School of Computer Science, McGill University

COMP-512 Distributed Systems

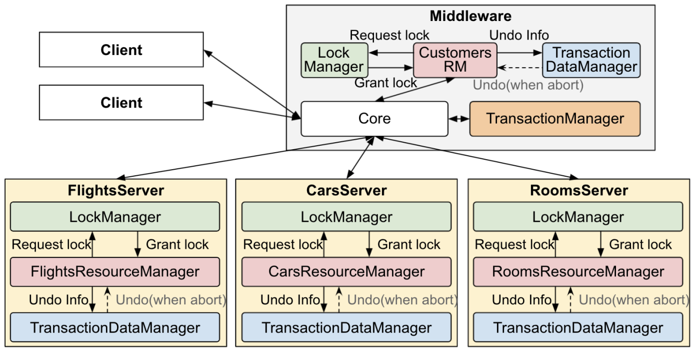
Project 2: Transactions and Concurrency Control

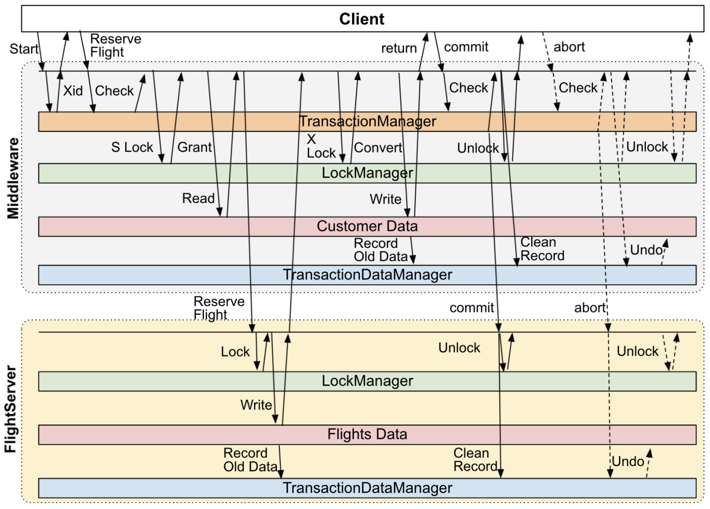
Group04: Elvin Zhang - Jack Hu - Mian Hamza

Part 1. Implementation

Distributed Transaction System Architecture

The Transaction Manager is part of the Middleware. It maintains a list of active transactions, TTL timestamps for active transactions, and a map of transactions and RMs that are involved in each transaction. Each server(including Middleware and 3 resource servers) has a Lock Manager that manages data locks, and a Transaction Data Manager that records database logs for undoing the changes when transactions abort.





Lock Manager

Lock Conversion

In the LockConflict() method, when a write lock is required by a transaction, and an existing read lock of the same transaction is found, we mark the bitset as 1. Then back in the Lock() method, if there is no other lock conflict, i.e. no other transaction holds a read lock on the same data, we should convert the existing read lock into a write lock.

Lock() and UnlockAll()

In servers, when we need to read data or write data(e.g. when reading customer data in the Middleware or writing flight data in the Flights Server), we call Lock() and use the Item Key as the data name.

DeadLock

When a lock waits for DEADLOCK_TIMEOUT(10 seconds) and cannot be granted, a deadlock exception will be thrown. When we catch a deadlock exception in readData() or writeData(), we inform the Middleware and abort the transaction.

Transaction Manager

The Transaction Manager is integrated in Middleware. It assigns a transactionId for a transaction when it starts, and keeps track of the states of transactions. When a client executes an operation of a transaction, it first checks if the transaction is active, committed, or aborted. It also records which resource managers transactions get involved. When the middleware needs to call a function on a resource manager, it records it to the transaction. When a transaction commits or aborts, it informs the involved resource managers to commit and abort. In order to implement the

Time-to-live mechanism, it also maintains a timer for each active transaction. Once an operation is called on a transaction, it cancels the previous timer and restarts a new timer. When timeout, it triggers the abort process.

Transaction Data Manager

The Middleware and the 3 Resource Servers all have a Transaction Data Manager. It maintains a hashmap that records previous values of data items each transaction changes. When a transaction writes new data, it stores the previous value according to transactionId and data item key. When a transaction aborts, it undoes the changes the transaction made.

Start

The start() function in the TransactionManager takes care of allocating a new transaction ID whenever a client starts a new transaction. It returns the xid field back to the client from the Middleware and adds a new TransactionObject() with it's xid to storage. We need to keep track of all active transactions. We also further add the TransactionObject to another storage where it keeps track of all resource managers accessed in it's active life-time.

Commit

The commit() function is implemented at the middleware and at each of the resource managers within the system. When a client calls commit for a transaction containing a certain xid, it sends the request over to middleware where the TransactionManager checks to see which resource manager (including the middleware itself) the transaction has accessed in its lifetime and which locks to unlock. It then sends commit requests to each of the resource managers to commit the changes made by that transaction. It returns false if unlocking all locks failed. Finally before returning true, it removes itself from the list of active transactions.

Abort

The abort() function is implemented at the middleware and at each of the resource managers within the system. When abort is called, it sends the request with the desired transaction xid over to the middleware. In the middleware it checks to see which resource manager the transaction has modified and for each of them, it performs data-undo protocol in order to write back to the DB the original values before operations changed it. After the data-undo protocol has finished for each resource manager, it then unlocks all the locks held. Finally it removes itself from the hashmap of active transactions and the hashmap containing each transaction's accessed resource servers.
Besides the client can call abort(), when a transaction has a deadlock, the transaction passively aborts. When a lock waits until timeout, the thread throws a deadlock exception. The Middleware informs all resource managers the transaction got involved and aborts the transaction.

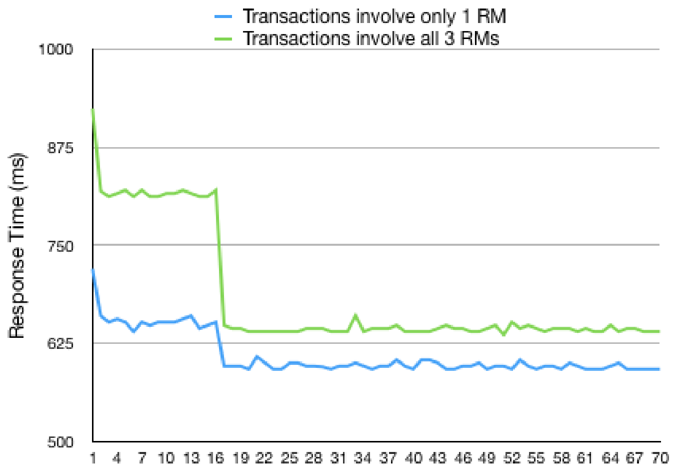
Part 2. Performance Analysis

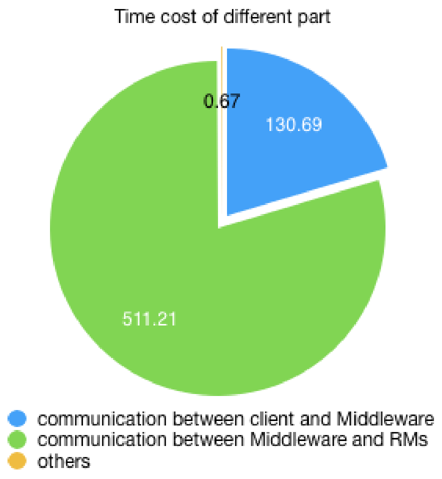
Test Client Implementation

We added several transaction types for our experiments. **AddQueryFlight, AddQueryCars, and AddQueryRooms,** each involves only 1 RM, and has 6 operations: start a transaction, add a customer, add an item(flight/car/room), query the item, query the price, and commit the transaction. **ReserveAll** involves all 3 RMs, and also has 6 operations: start a transaction, add a customer, reserve a flight, reserve a car, reserve a room, and commit the transaction.
We also added a new command **ExecuteTestSuite** that manages and runs multiple transactions of a specific resource ("homo" mode) or a mix of multiple resources ("hetero" mode) by a loop. We are able to set the throughput (transactions/s), and the client can choose between a short, medium, and long number of operations for each transaction. After each transaction is committed, we record the response time. If the response time is less than 1/throughput s, we need to sleep the thread for (1/throughput - response time) seconds. To ensure that there is a

variation between the different transaction times of $\pm |x|\ ms$. We generate a random number less than 10% of wait time and add the variation to the wait time.
Our test commands also record the startTime and endTime for each transaction into a clientLogger. The clientLogger then writes the startTime, endTime into a Hashmap. Once all of the transactions are completed, the clientLogger is dumped into a CSV file, this occurs by writing the contents of the Hashmap. On the server side, we also record the DB time, function execution time, and then calculate the communication time between servers and clients.

Experiment 1 - Single Client:



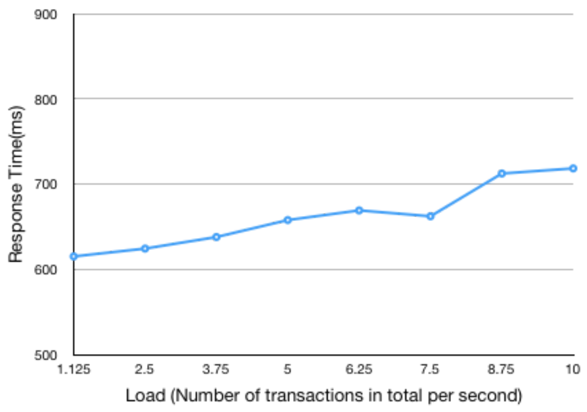


We run **AddAndQueryCars** and **ReserveAll** on a client. Both transactions have 6 operations as mentioned before. According to the experiment result and the graphs provided, we can observe:
(1) The response time is higher at the beginning and becomes stable after a while. This is because at the beginning the system spends some time on initialization.
(2) After becoming stable, the average response times of these 2 transactions are about 595.35ms and 642.57ms. The average response time for transactions involving all RMs is greater than only 1 RM. This was to be expected as the operations involved would request 3 different resource servers and increase the communication time involved.
(3) Both the homogeneous and heterogeneous transaction types spend most of their time in request/data communication between the underlying servers involved. The average execution and "DB" time are negligible for the resource servers.

Experiment 2 - Multiple Clients: **(In experiment 2, we modify the DEADLOCK_TIMEOUT to 2000ms.)*

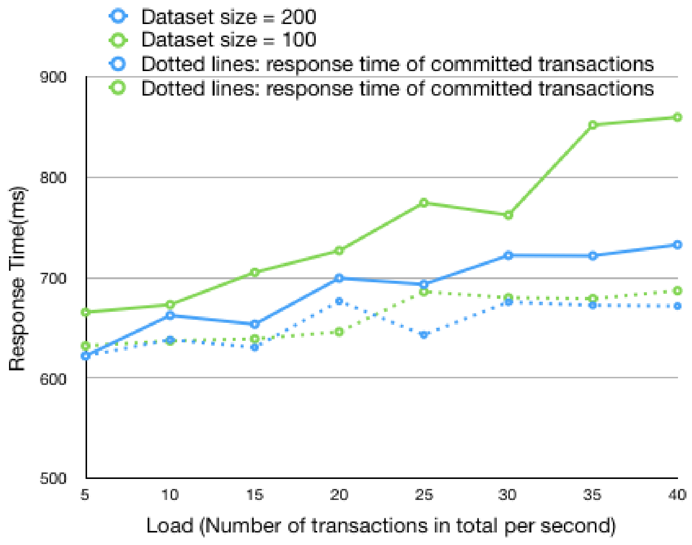
5 Clients:

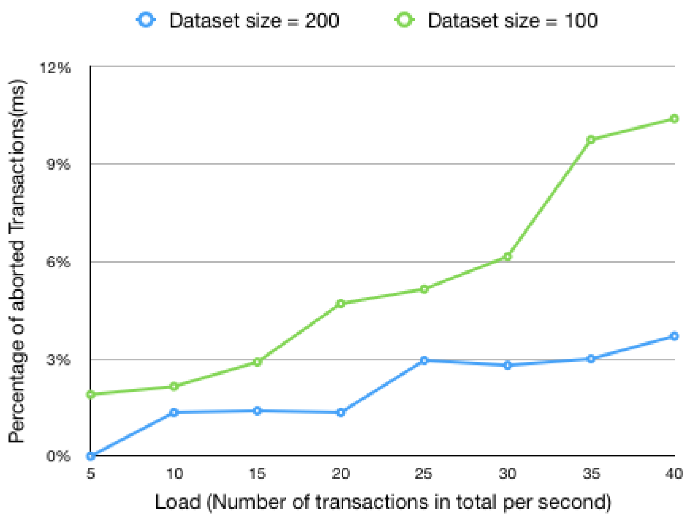
For this experiment, we run **ReserveAll** transactions on 5 clients at the same time, with different throughput settings. The graph represents a 5 client system where the throughput is increased from 1.125 transactions/s to 10 transactions/s (the number of transactions/s/client is increased from 0.5 to 2). As one can see that when the throughput was increased, the response time of the system increased as well. This was to be expected as the servers need to handle more transactions at the same time, and there will be more opportunities for lock conflicts that will cause wait time.



20 Clients:

For this experiment, we run **ReserveAll** transactions on 20 clients at the same time, with different throughput settings and dataset size settings. In order to figure out the effect of the lock conflicts on response time, we run the experiment with different sizes of dataset. First we set the dataset size as 100, i.e. there are 100 flights and 100 locations for cars and rooms, and each transaction randomly picks 1 flight and 1 location among the whole dataset and reserves them. Then, we set the dataset size as 200. There will be more flights and locations. The probability that 2 transactions pick the same item to reserve at the same time will be lower, so the number of lock conflicts will be lower.





According to the experiment result and the graphs provided, we can observe:

(1) For both sizes of dataset, the response time increases as the throughput increases, because servers need to handle more transactions at the same time and there will be more lock conflicts that cost time.

(2) The percentage of aborted transactions due to deadlocks increases as the throughput increases, because there will be more lock conflicts when more transactions come at the same time. When the size of the dataset is 200, the percentage of aborted transactions is lower, because when there are more items, the probability that 2 transactions pick the same item to reserve at the same time will be lower.

(3) When the size of the dataset is 200, the response time is lower than when the size is 100. This is consistent with the data of percentage of aborted transactions. It is also because when the dataset is bigger, the probability that transactions request the same data is lower, so the lock conflicts and deadlocks will be more.

(4) We also calculated the average response time of committed transactions(the 2 dotted lines). As we can see, it also increases as the throughput increases, but the gap between 2 dataset size experiments is less. It means the concurrency control is a bottleneck when the data size is 100.

In conclusion, when there are 20 clients, and the data size is 100, the concurrency control is a bottleneck of the system.

Contribution

Elvin Zhang: 1. Implement lock conversion, lock mechanism, and deadlocks handling.
2. Design the architecture and implement the distributed transaction system.
3. Implement transaction management, start, commit, abort, shutdown on servers, Middleware and client side.
4. Implement TTL-mechanism, time recording on server side.
5. Design and run experiments on a single client, 5 clients and 20 clients and analyze the result.

Jack Hu: Implemented first version of TestClient which supports all the performance test commands submitted to the reservation system. Implemented throughput, parametrized transactions, and client transaction logging functionalities. Help refactored and clean up code related to concurrency control.

Mian Hamza: Implemented the TestClient functionality that deals with creating functions for the transaction. Wrote the functionality for the throughput, and created a technique to ensure that random throughput variation exists between different transactions.