Performance Analysis

Architecture

To conduct the performance analysis, I modified the existing code to implement timers, to measure the response time. The new architecture is shown in Figure 1. To produce the data, several tests were executed and various times were returned:

- Pick transaction type (single Resource Manager or all Resource Managers), client number, throughput
- Spin up n threads (n clients) and connect these clients to the middleware
- Before each transaction, record the $beforeTime_{RT}$; after the transaction record $afterTime_{RT}$. Thus, RT = $afterTime_{RT}$ – $beforeTime_{RT}$, which is the total time for the transaction to be committed. Thus, RT includes: communication time, middleware time, resource manager time, and DB time
- At the Resource Manager, record the $beforeTime_{RM}$ and $afterTime_{RM}$. Thus, RM includes the DB time. Return the RM = $afterTime_{RM}$ – $beforeTime_{RM}$ and the time it takes to read/write the global m_data to the middleware.
- At the middleware, record the $beforeTime_{MW}$ and $afterTime_{MW}$. For MW, also subtract out any time waiting for the resource managers. Thus, MW is the total time at the middleware. Return [MW, RM, DB] to the client.

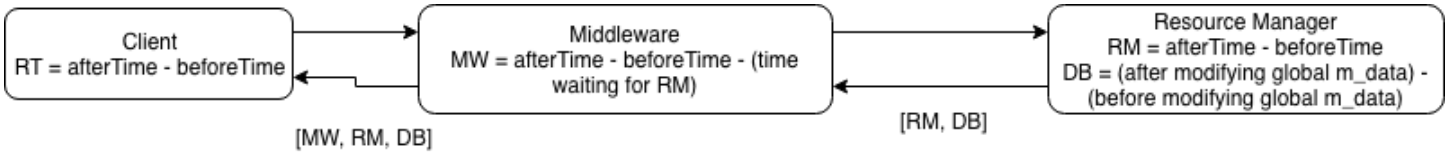

Figure 1

Test Bed

For all tests, the distributed environment consisted of 3 Dell Computers (in the Trottier 3rd floor lab), which each have: 7.7 GB of memory, Intel Core i7-4790S CPUs, and are running Ubuntu 16.04. 1 computer runs the client(s), 1 runs middleware, and 1 runs the 3 resource managers.

Each test consisted of 100 Flights, 100 Cars, 100 Rooms, and 500 customers. Thus, deadlocks can occur, which will result in timeouts. After every test, the environment is reset. For each client, 150 transactions are put through. The first 100 are thrown away (to ensure the caches aren't empty) and 50 response times (per client) are kept for analysis. For the multi-client tests, 5 concurrent clients were used. Moreover, after each transaction, the client waits x ms, to ensure the proper throughput. To add variation, I added ±30 ms (thus, each client waits y ms according to the interval [x – 30, x + 30]).

The two transaction types are as follows (Note: clients choose customer ids, flights, cars, and rooms to reserve randomly):

Single Resource Manager
- *Start()*
- *queryFlight()*
- *queryFlightPrice()*
- *reserveFlight()*
- *commit()*

All Resource Managers
- *Start()*
- *reserveFlight()*
- *reserveCar()*
- *reserveRoom()*
- *commit()*

As for measurement techniques, refer to the Architecture section above. (**NOTE: ResponseTime = RT, MiddlewareTime = MW, ResourceManagerTime = RM – DB, DatabaseTime = DB, CommTime = RT – MW – RM)**

Performance Figures – Single Client

Response Time (ms)	Middleware Time (ms)	Resource Manager Time (ms)	Database Time (ms)	Communication Time (ms)
10.14	2.9	0.38	0.06	6.8

Figure 2 - Single Client Average Response Time for single resource manager

Response Time (ms)	Middleware Time (ms)	Resource Manager Time (ms)	Database Time (ms)	Communication Time (ms)
13.4	3.42	0.58	0.24	9.16

Figure 3 - Single Client Average Response Time for all resource managers

For the single client tests, 2 transaction types (see *Single Resource Manager* and *All Resource Managers* transactions on the previous page) were executed 150 times, the last 50 response times were kept, and these response times were averaged. In both cases, communication time dominated the total response time. With one client, this makes sense, since the middleware is coordinating sequential (no concurrent) transactions (thus, middleware time is quite small and reasonable).

Performance Figures – Multi-Client (n=5)

For the multi-client setting, 5 clients were used and 6 tests were used with throughput (trans/sec) = 1,10,25,50,100,200. For each transaction type, each client was executed 150 times (so, 750 transactions in total), and the last 50 transactions of each client were kept (thus, averaging was over 250 transaction response times). Referring to Figure 4, the single resource manager transaction type, the saturation point seems to be around 25 trans/sec. Before this point, response time averages around 9 ms. After this point, there is a significant increase (> 100 ms) in response time. Reviewing the data (found in the data/ folder) for Test 3 – Test 8, it reveals that the increase is due to middleware time (with a few deadlocks).

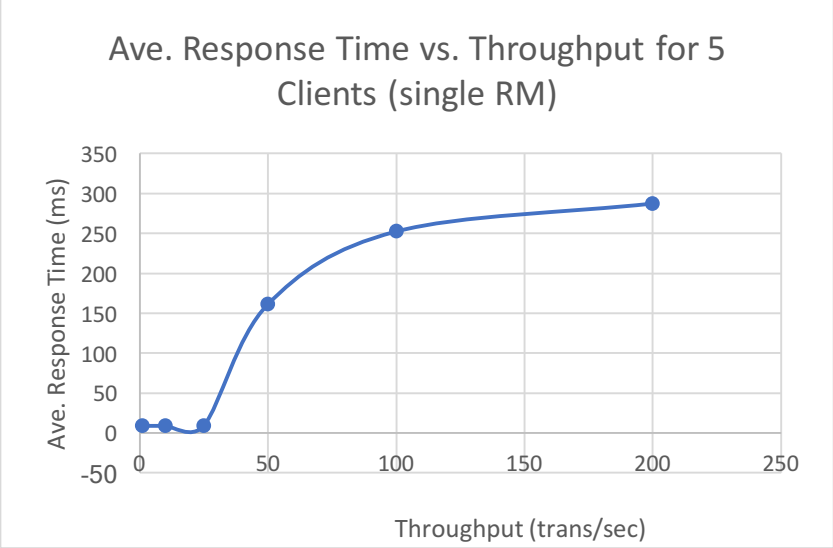

Figure 4

The Middleware, in my implementation, coordinates the execution of the resource managers **and** stores the Customer resource manager. Thus, this is a significant bottleneck, as one server is relied on to both handle *Customer* data and coordinate the activity of the other resource manager servers.

For the *All Resource Managers* transaction type, the execution is exactly the same as the *Single Resource Manager* (the only difference is the operations executed). Referring to Figure 5, the saturation point, again, seems to be around throughput = 25 trans/sec. Reviewing the data (Test 9 – Test 14), again the Middleware seems to be a bottleneck, as the increase in throughput strains this resource. This transaction type relies on the *Customer* data and the coordination of the 3 Resource Managers, and so, the middleware would be under a lot of strain during a high throughput time. As such, the average response time would increase.

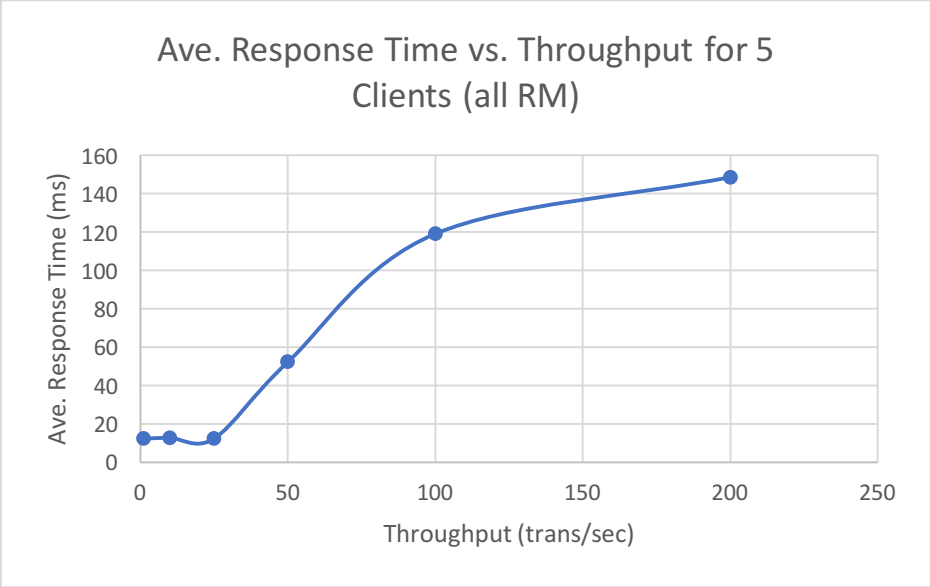

Figure 5

Custom Functionality

Since Group 7 consists of only one person, Alex allowed me to skip the custom functionality requirement.