PG - 1
COMP-512: Distributed Systems, Fall 2017

Project Phase 2: Locks & Trasaction Management

Shabir Abdul Samadh
McGill University
Student ID: 260723366
shabir.abdulsamadh@mail.mcgill.ca

Jianhua Li
McGill University
Student ID: 260351181
jianhua.li2@mail.mcgill.ca

Keywords

Distributed-Systems; Middleware; Transaction Management; RMI; 2-Phase Locking; Concurrency-Control

1. OVERALL FLOW OF A TRANSACTION

Unlike before, if a client wants to read or write an item, the client has to start a new transaction. And after finishing the desired operations, the client has to commit the transaction in order to save all the changes. Each time a new transaction is called, we create a new transaction object. Inside the *transaction* class, there is a map *accessedItemSet* that contains all the data items that have already been accessed.

2. TRANSACTION MANAGER

2.1 Reserve Operation

When we call *submitReserveOperation(tId, customerItem, requestType, resourceItem)*, first we retrieve the *Transaction* object. Then we have to acquire the necessary lock for the operation. After getting the lock, first we check whether the customer has already been accessed within the same transaction. If so, we just directly use the customer object that's inside the *Transaction* object. If not, then we have to start a mini transaction where we try to get the customer object from customer manager which is inside the middleware, once we get the *customer*, we commit this mini transaction. Then we need to actually reserve the item for the customer, for which we have to call *submitOperation(tId, requestType, resourceItem)* first. Inside this method, it will also first check whether the item has already been accessed by looking inside *transaction.getAccessedItemSet()*, only has it not been accessed will it try to retrieve the reservable item from the resource managers. Eventually it returns the desired value so that we can proceed with the reservation.

2.2 Commits

When committing, for each of the resource type, we start up a transaction with the appropriate resource manager. Then for each modified item, we use the appropriate action (write or delete). After we finish all the actions we just commit the transaction. And after we are done with all the changes, we unlock all the locks held by the transaction.

3. PERFORMANCE ANALYSIS

In terms of analyzing the performance of this system we considered two specific cases - load produced by *a single client* & by *many clients* concurrently. The performance metric that we were specifically targeting was the **R**ound **T**rip **T**ime *(RTT)* of the transactions introduced by the client(s). In addition our analysis was setup in such a way that the following scenarios were test for:

- *Read* operations on a single **R**esource **M**anager (RM).
- *Read* operations on multiple RMs.
- *Write* operations on a single RM.
- *Write* operations on multiple RMs.
- Transactions with a combination of *Read & Write* operations on a single RM.
- Transactions with a combination of *Read & Write* operations on multiple RMs.

For the case of testing the system on **multiple-concurrent-clients**, the transactions produced per client was a random mix of *Read & Write* operations.

3.1 Setup & Results

For the test scenarios given above, the *Read* operations were **randomly** picked on each iteration to be either *queryResource* or *queryResourcePrice*. For the write operations, they were picked to be one out of *addResource*, *deleteResource*, *reserveResource* or *reserveItinerary* at each iteration. Also for each test case some randomly generated resources were added to the RMs to ensure they are not empty during the test. Identifiers to the randomly generated items were kept locally by the **analysis-client**. The client randomly picks an item from this list to carry-out one of the operations discussed above. Also, the client was setup such that it randomly picks on which RM the operation is going to be performed in random. Thus, multiple runs of the *RTT* analysis would not result calculations biased to a single RM.

For the last 2 *(out the 6)* listed above, the client generates a random number between 1 and 10, which will denote the number of operations per transactions. Subsequently for each of the operation in the chosen number of cycles, the client randomly decides to do a *Read* or a *Write* operation. Finally, based on this decision it picks the operation also in random as mentioned above. For the *single client* scenarios we run each test for 100 loops and we vary the test by varying the transaction per sec **(load)** on the system. For the *distributed client* setup, we let each client submit 20 transactions *(per test)* on an interval decided by the load to be submitted to the systems. We also varied the number of clients between 10 and 40, to identify the impact of increased concurrent clients.

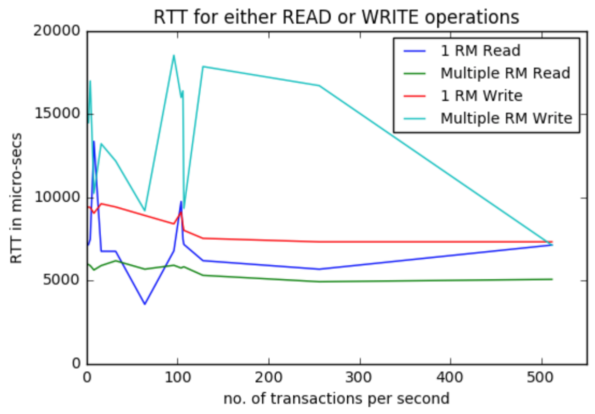

Figure 1: RTT for either Read or Write operations

The figure above shows the RTT results for the 1st 4 cases listed earlier. It is noteworthy that the average RTT for *Writes* are higher compared to *Read* operations. This is directly intuitive by the fact that for *write* involved commits the *middleware* must contact the corresponding *RM*, whereas *reads* does not require the *middleware* to do any further processing. It addition we can also conclude from the above results that continuous *reads* on a single *RM* seems to be taking longer to respond that *reads* on multiple *RMs*. We believe that this primarily given that the load that each *RM* has to handle is distributed with *multiple-RM-reads* compared to *single-RM-reads*. However, this is not the same for the case of *writes*; *multiple-RM-writes* takes more **RTT** than *single-RM-writes*. This can be explained by the fact that with multiple *RMs* involved the proxy-stub invoked by the *middleware* is continuously changing for subsequent requests. Hence, we believe this does not allow the *JVM* to optimize the calls efficiently compared to the case of a *single-RM-invocation*.

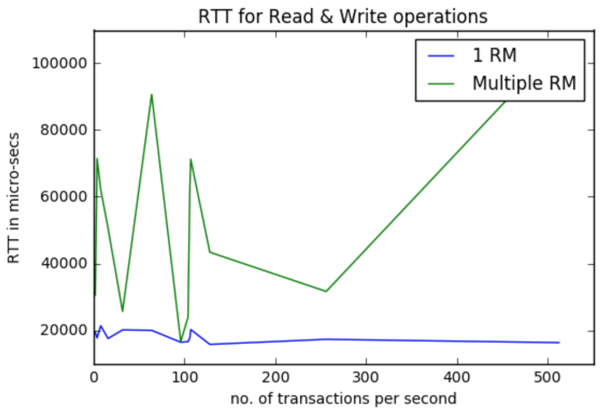

Figure 2: RTT for Read+Write operations

Figure 2 shows the **RTT** variation for transactions involving a random set of *Read & Write* operations. It was tested for two cases: *Read+Write* operations on (1) a single *RM* and (2) on multiple *RM*s. We can see that the average **RTT** is much higher when multiple *RM*s are involved. We can explain this by a similar deduction as before: multiple *RM* based transactions require interactions to varying *RMI-registry* **proxy-stubs**. This weakens the level of optimization that the *JVM* could provide to the system. However, with a single *RM*, the calls fall within a specific set of elements and the system could support this much faster & efficiently.

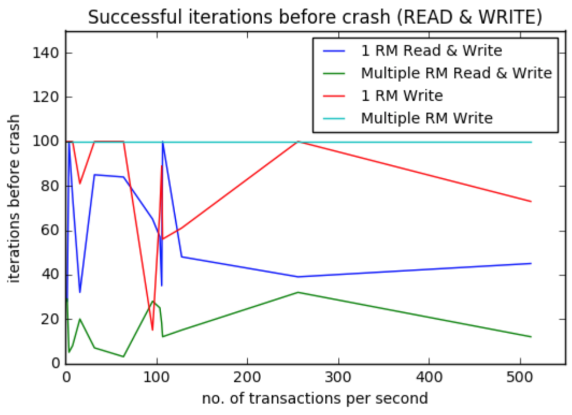

Figure 3: Successful iterations (Write & "Read+Write" operations)

Figure 3 is a depiction of *loop-counts* before **crash** *(or exception scenario)* for 2 cases discussed above: (1) *Write operations* and (2) *Read + Write operations*. It can be noted that the success-rate `(without failure)` is higher in the case of transactions involving only *write* operations compared to *read + write* combined transactions. Whilst this can be pretty intuitive by the fact that a combinations of operations will definitely result in a longer commit period. However, what is interesting here is that for *writes-only* test, *multiple RMs* seems to be handling crashes better than *single RM* scenario. On the other-hand for the *read-write combined* test, a *single-RM* performs better than with *multiple-RMs*. We believe to have introduced some sort-of error into our test resulting in what we observe.

Table 1: Concurrent Clients invocation

Load	10 clients	40 clients
1 t/sec	36 ms	58 ms
2 t/sec	33 ms	48 ms
10 t/sec	28 ms	22 ms

Table 1 shows our results for **RTT** for transactions when *concurrent-clients* produced load. We see that with increasing no of clients the **RTT** increases. However, we also note that for a fixed client number, increasing load sometimes reduces **RTT**. We describe this as an error we made in adding a `ZERO` to the average calculation for iterations throwing a `DEADLOCK`.