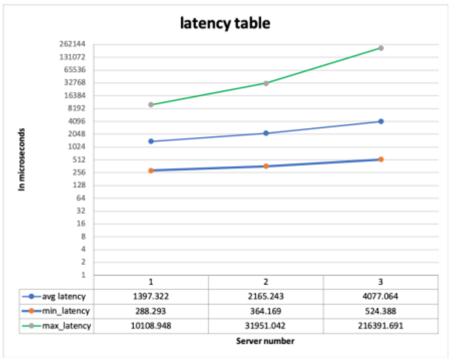
# Assignment 2 report

In the first part of the assignment, I implement the customer records using a map. The key of the map is the customer id, and the value is the last order number of the customer. We also need a state machine log to keep track of our operations of the customer records. I firstly build the MapOp using struct which contains the opcode, customer id (arg1), and parameter for the operation(arg2). The state machine replication log is a vector of MapOp. I create and update some data formats listed in 1.2.2 sections followed by instructions in the assignment doc. To implement the customer-record-access workflow, I mainly change the expert thread to an admin thread and updated how engineer thread send admin update requests. An admin update request is implemented as struct that contains the new customer record, and a promise of the admin id (for engineer threads wait for the admin finishing the update). For robot manufacturing workflow, the engineer will fill the robot as usual, then it will push back an admin request with a new customer record to the admin request queue and notify the admin thread. The admin thread will receive the notification and pop the admin request from the queue and applied MapOp in the request to the customer records. Then it will return the admin to notify the engineer the update is done. For customer record reading workflow, the engineer will check the request type of the customer request. If it is a record read request, the engineer thread will lock first the customer records lock and read the corresponding record from the customer record map and send it to the customers. For 1.3 and 1.4 sections, I mainly worked in client Main to parse the three types of request and the client thread. For request type 3, one client thread will be created and send as many as # orders read requests to scan all customer records. The request type 3 is only known to the client side. Client will only use type 2 for sending read request.

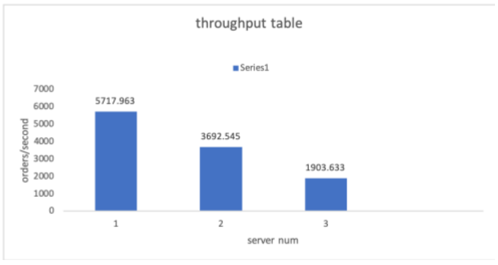
For the second section, we need create multiple server nodes for a primary factory and idle factories. First, I updated the parser in server main to correctly parse information of its peer nodes and also its own factory id and port number. I store all these data in a struct called admin_config, and admin_config is stored in the factory class; since these are all information needed by admin thread. The admin_config contains the last_index, commited index and primary_id and its unique id, and a map of its peer nodes (key is peer id and value is a Peer). To further represent and store data for peers, I created a Peer class which contains peer's unique id, peer's ip and peer's port. By this, the primary node can easily access to its peers' data. Since, we need communicate between peer nodes. I created some necessary communication classes and add some messages, including FactoryStub, FactorySocket (identical to ClientSocket, and it is for solving the compile issue), IndentifyMessage, ReplicationRequest and MapOp. The Identification message is used for distinguished if the connections are built for client with flag of 0 or peer nodes with flag of 1. The ReplicationRequest class contains all necessary information for replication, which are factory_id (where the request is sent from), primary's commited_index, primary's last index and the MapOp. As I mentioned before, I built a class for MapOp, since it was involved in network transactions and I need handle marshal and unmarshal for it. FactoryStub will help for initializing peer connections and peer connections will then store in a map (key is peer id, value is std::unique_ptr<FactoryStub>) in the factory class for easier access. Once the connections are created between primary and idle factories, Identify Message and then Replication Request can be sent. Moreover, I update ClientStub to make it be able to send identify message as the client and ServerStub to receive identify message and replication requests. With all these implemented, the new workflows work now. The engineer threads will try to receive identify message. If the identify message was sent by client, it will do as usual as an engineer. If the identify message is sent by PFA, it will work as IFA followed by instructions to update its log and customer record so that customers can also read data from IFA.

For the third part, we need consider nodes failures. To represent a node is failed, I add a bool status in the Peer Class. Once the Peer nodes (could be either IFA and PFA) failed, I will shut the peer connections down and set peers' status to false. PFA will check if its peers are down before sending identity message and replication requests. If peers are down, it will skip it. If we switch PFA, it will try to connect peers, if connections are failed, it will also shut the peer down by setting the status to false. For my design, I can't find a place to set the primary id to -1, since the engineer thread will receive failure message as -1 from both clients or peers and it can't really figure out where they are from. However, with this bool status, it works fine. Moreover, there are some cases that the last index, committed index and the customer record messed up because of nodes failures. Therefore, I check if I can safely pushback or I need insert in certain index to fix index issues.

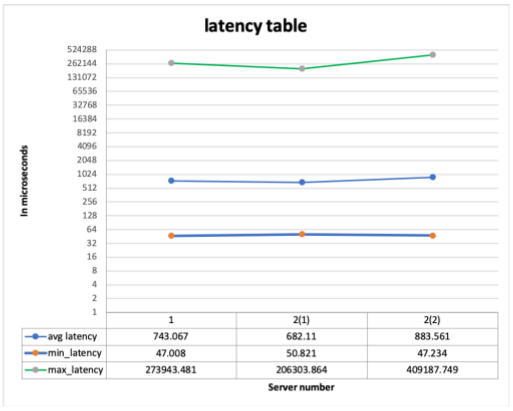


| | 1 | 2 | 3 |
|---|---|---|---|
| avg latency | 1397.322 | 2165.243 | 4077.064 |
| min_latency | 288.293 | 364.169 | 524.388 |
| max_latency | 10108.948 | 31951.042 | 216391.691 |

This is latency table for 1 to 3 servers and 8 engineer threads which make different number of orders. As we can observed that all three latencies are increased when there are more server nodes. This makes sense since it takes more time for more replication being made in more server nodes.

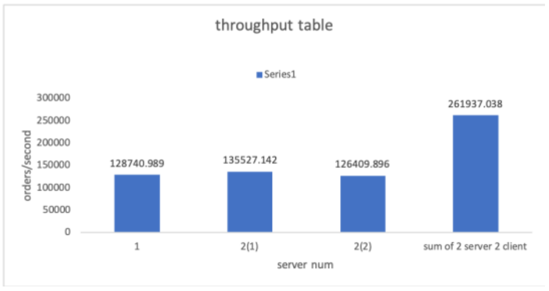


This is the throughput table for 1 to 3 servers and 8 engineer threads which make different number of orders. As we can observed that the throughput of 1 server node is the highest, 2 servers becomes a lot less, and 3 server becomes even more less. This also makes sense, since times are spent on making more replications for more server nodes.



| | 1 | 2(1) | 2(2) |
|---|---|---|---|
| avg latency | 743.067 | 682.11 | 883.561 |
| min_latency | 47.008 | 50.821 | 47.234 |
| max_latency | 273943.481 | 206303.864 | 409187.749 |

This is the latency table for 1 server with 1 client and 2 servers with 2 clients (1 client each). As we can observed that all three types of latencies from different situation vary not too much, except that the maximum latency of one of 2-clients case really high. This makes sense, since reading requests can be processed identically by primary server node or idle server nodes. The reading process of 1-server-1-client is as same as reading process of 1-primary_server-1-client or 1-idel_server-1-client. The results from these two settings should be similar.



This is the throughput table for both 1 server with 1 client and 2 servers with 2 clients (1 client each). As we can see, the throughput is similar for both situations. The sum of 2 clients' throughput are 2 times of 1 client's thoughput.