ASSIGNMENT 4

DISTRIBUTED KV STORE

DESIGN DETAILS AND TESTS:

SERVER: The application contains 2 identical TCP Servers built using python3 which handle connections from clients and performs read and write operations for the commands being sent in the key-value store. For the scope of this project, every incoming request provides the consistency model which is to be used to process it in order to be replicated to the duplicates. The server checks if all the command protocols are being followed properly and responds with equivalent error commands in case of any ill formed command being sent. To start the servers a script can be run which has a syntax as follows:

./start_servers.sh

In all the design protocols, server 1 acts as the primary server and handles all the write requests and all the read requests are being handled by server 2. I have implemented the sequential consistency and linearizability using a decentralized total-order-broadcast. For each incoming request, I have added a flag which tells the server as to which consistency scheme to follow in order to handle the current request. All the shared variables among the servers and their required configurations are defined in ConfigFile.py. The implementation description, of each consistency model is as follows:

EVENTUAL CONSISTENCY:

In the case of eventual consistency, all the read and write operations take place directly as a separate process on the respective servers. For this, the flag bit holds information as 'eventual'. Further when any server receives a write request, after updating the value in it's local copy it completes the transaction with the client closing the connection between them and then broadcasts it to server 2, so that consistent copies of the databases can be maintained. Since both the servers are running in the local system there is very less propagation delay between them and thus it is difficult to observe with tests. Due to this, I have added a delay of 10 seconds between when the value is stored in the receiving server and when it triggers a broadcast to the other server. Both the read and write operations are not blocking in this case and every request is processed as it is received in a multi-threaded server structure.

Tests to show Eventual Consistency:

To run the tests for eventual consistency, run the below script:

./start_eventual_test.sh

In these test files, the first client sends a write request to the server and changes the value of key "test1". Then I send a read request to server 2 and it can be observed that it still fetches the stale value. To confirm, that the value was updated I send a read request to server 1 and it shows that the value is indeed updated for the primary. Then after a wait of around 10 seconds, I try to read the value again for the same key from server 2. This time, it fetches the updated value of the key from the same server. Thus, assuring that eventually a consistency between the two servers is achieved.

The test results for eventual consistency are as follows:

```
**********************************************************************
TEST FILE TO CHECK FOR EVENTUAL CONSISTANCY
**********************************************************************

Starting client to Set value for key "test1" in server 1

STORED

Ending connection for Set Operation

**********************************************************************

Total time for write operation in eventual consistency: 0.775120735168457
Starting client to Get value for key "test1" from server 2

VALUE test1 50
1
END


Ending connection for Get Operation

Total time for read operation in eventual consistency: 1.5677931308746338
**********************************************************************
```

```
Starting client to Get value for key "test1" from server 1

VALUE test1 51
15
END


Ending connection for Get Operation

**********************************************************************

Starting client to Get value for key "test1" from server 2 after eventual consistancy is achieved

VALUE test1 51
15
END


Ending connection for Get Operation

**********************************************************************
```

SEQUENTIAL CONSISTENCY:

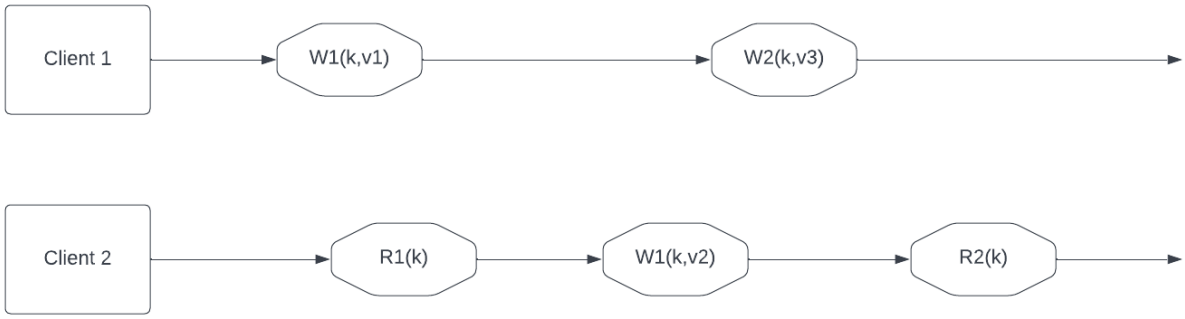
Sequential consistency is implemented using the local-read algorithm. All the reads are local and write operations are blocking. For this, the flag bit holds information as 'sequential'. When any server receives a read request, it directly serves the client value of the required key from it's own local storage irrespective of the fact that any write is being executed. When write request is received, the server first updates it's local value to the requested write and then broadcasts the same write message to all the other servers (in our case only to server 2). It holds a lock over the execution of any other write operation which means, all the other write operations requests are made to wait unless broadcast response from server 2 is not received. If during a write process in this case, a read request is initiated to the server 2, it will show stale value unless it's from the same client in which case the updated value is returned as it will only be executed only when the first operation is completed which here is a write, thus conserving the happens before relationship for a process.

Tests to show sequential consistency:

To run the tests for sequential consistency, run the below script:

./start_sequential_test.sh

In these test files, the requests from the clients can be understood in the below diagram,



First client sends a write request (W1('test1', '1')) to the server 1 and changes the value of key "test1". Then from client 2 send a read request to server 2 and it can be observed that it still fetches the stale value 3 from test1 which shows that from a global point of view R1(k) from client 2 is the first command to be completed. Then although W1(k,v2) contacts the server, it is still not executed by the server as write from server 1 is still running. Once it gets the lock it updates the value from its current lock and only then can W2(k,v) from client 1 can proceed. During this, the server also receives R2(k) from client 2 and since it is a local read, it gives the stale value of '3' which is now updated from W1(k,v) from client 2. Thus sequential order of each client is maintained and it can be seen that the order of operation from different clients is changed among them.

The test results from sequential consistency are as follows:

```
**************************************************************************
TEST FILE TO CHECK FOR Sequential CONSISTENCY
**************************************************************************

Starting client for W1(k,v) from client 1

Operation starting at: 19:39:32

Starting client for R1(k) from client 2

Operation starting at: 19:39:32
VALUE test1 50
1
END


Ending connection for R1(k) Operation from client 2

Total time for read operation in sequential consistency: 1.5008373260498047
Starting client for W1(k,v) from client 2


Ending connection for W1(k,v) Operation from client 1
Operation starting at: 19:39:37

**************************************************************************
```

```

Starting client for W2(k,v) from client 1


Operation starting at: 19:39:42
Ending connection for W1(k,v) Operation from client 2

**************************************************************************

Total time for write operation in sequential consistency: 4.8155882358551025

Starting client for R2(k) from client 2

Operation starting at: 19:39:45
VALUE test1 50
1
END


Ending connection for R2(k) Operation from client 2

**************************************************************************
```

LINEARIZABILITY:

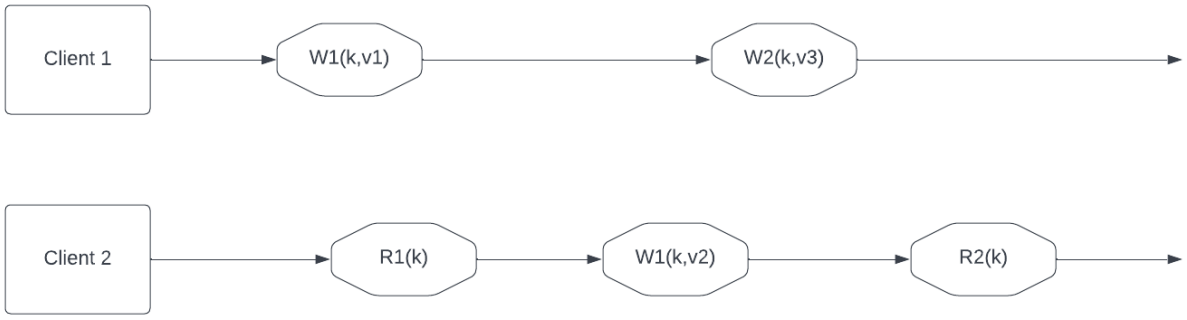
Linearizability is implemented over the sequential consistency algorithm. All the read and write operations are blocking. For this, the flag bit holds information as 'linear'. When any request is received by the server, it attaches a sequence number to that request, be it read or write, and holds an internal counter shared by all the servers, to know which sequence number is last executed. When an operation is completed, only then the request from the next sequence number is served. For linearizability, the only difference is that even though reads are still local, they are also assigned a sequence number and are only executed when their respective sequence number is achieved. This means that there are no stale reads in this process as a common global perspective of operations is achieved.

The test results from linearizability are as follows:

To run the tests for sequential consistency, run the below script:

./start_linear_test.sh

In these test files, the request from the clients is the same as in the case of sequential consistency to show the difference between both the processes,



When these test cases are run for linearizability, we observe that the order of operation is as follows:

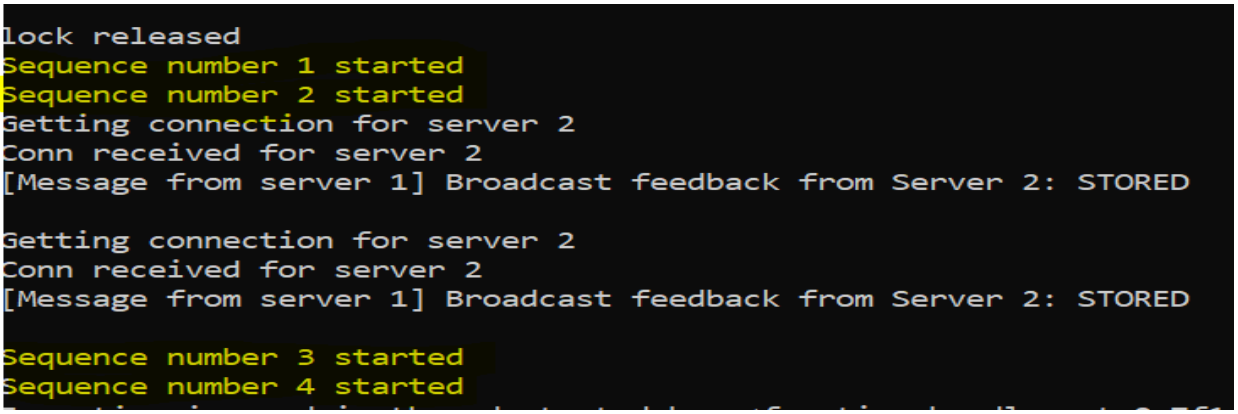
W1(k,v1){Client 1} ► R1(k){Client 2} ► W1(k,v2){Client 2} ► W2(k,v3){Client 1} ► R2(k){Client 2}

It can be further verified with the below logs,

```
**********************************************************************
TEST FILE TO CHECK FOR Linearizability
**********************************************************************

Starting client for W1(k,v) from client 1

Operation starting at: 18:55:26

Starting client for R1(k) from client 2

Operation starting at: 18:55:33
STORED

VALUE test1 50
1
Ending connection for W1(k,v) Operation from client 1


END
**********************************************************************


Ending connection for R1(k) Operation from client 2

Total time for read operation in linear consistency: 2.0756995677948
Starting client for W1(k,v) from client 2

Operation starting at: 18:55:41
Starting client for W2(k,v) from client 1


Ending connection for W1(k,v) Operation from client 2
```

```
**********************************************************************

Total time for write operation in linear consistency: 3.2436888217926025

Starting client for R2(k) from client 2

STORED
Operation starting at: 18:55:48

Ending connection for W2(k,v) Operation from client 1
VALUE test1 50
1


**********************************************************************
END


Ending connection for R2(k) Operation from client 2

**********************************************************************
```

Another important thing to note here is that when I ran this code to normally check the working of the linear order, it showed that all the operations are executed in order. It can be confirmed from server logs as below:



PERFORMANCE EVALUATION:

For evaluating the performance of each consistency model, I have used execution time between triggering of request and completion of request for intermediate read and write requests. I believe this would be a good evaluator for latency as well as it will help in confirming if the models are following the algorithm properly.

Model	Time in Seconds	
	Write Operation	Read Operations
Eventual Consistency	0.77	1.56
Sequential Consistency	3.24	1.50
Linearizability	4.81	2.07

On the basis of latency, eventual consistency performs the best as it has the least operation time for both read and write operations and thus has the best availability among all the three models. But, the delay that was added to show eventual consistency will be removed in a practical distributed system and thus to get accurate reads the propagation time will have to be added. In sequential consistency, the reads are local and thus they take the same amount of time as in the case of eventual consistency. The write operations are blocking due to which the time taken to complete intermediate write requests is substantially more. For linearizability, the time for write and read requests are both higher as both read and write operations are now blocking. Due to this, even intermediate write operations will take longer for their turn and the intermediate read operations will also wait for the previous read/write request to be completed in order to be served.

Thus, we can conclude when latency is the deciding factor for model selection the order of consistency models in increasing order will be:

Linearizability < Sequential Consistency < Eventual Consistency

One tradeoff for this order is, that when we choose latency, the overall order of operations and in some sense, consistency is compromised. It can be showed as:

Eventual Consistency < Sequential Consistency < Linearizability

MAIN CHALLENGES:

- The major challenge was to decide on which algorithms to use for the case of sequential consistency and linearizability. I ended up using a decentralized total-order-broadcast approach for these consistency models.
- Different clients were behaving differently in terms of connection time so, implementing these models for the test clients was quite tricky. So, I had to add buffer times for starting some of the requests.
- Since, for some cases the requests are blocking, I had to make use of sharable locks between both the servers which caused a lot of problems and lead to race conditions in many cases. To overcome this, I had to make use of 2 locks for different applications.
- The size of the application increased from the simple KV Store and bug detection was getting difficult, so log statements for servers and test clients were important.