Aditya Arun Rudrawar

Distributed KV Store

HOW TO RUN

1. Import the kvstore file from the library
2. You can start the program by initializing the class like `kvs = kvstore.KVStore()`
3. Params that you can pass to KVStore

```
    consistency : string,
    replicas : int = 3,
    storage_directory : string = '',
    output_directory : string = ''
```

4. After you have initialized the object of the kvstore class. You can start the controlets and datalets by calling the `start()` method on the class
5. You can get the addresses of all the controlets get_controlet_address() method.
6. Clients can connect through the standard pymemcache client.

```
    from pymemcache.client.base import Client



    c = Client(address)
    response = c.set(key, value, noreply=False)
    print("SET response ", response)
    c.close()
```

7. You can pass the respective string for the consistency you want.

```
EVENTUAL_CONSISTENCY = 'eventual_consistency'
LINEARIZABLE_CONSISTENCY = 'linearizable_consistency'
SEQUENTIAL_CONSISTENCY = 'sequential_consistency'
```

8. You can see any of the test files to get an idea. Everything is set in env

CONSISTENCIES IMPLEMENTED

1. Linear
2. Sequential
3. Eventual

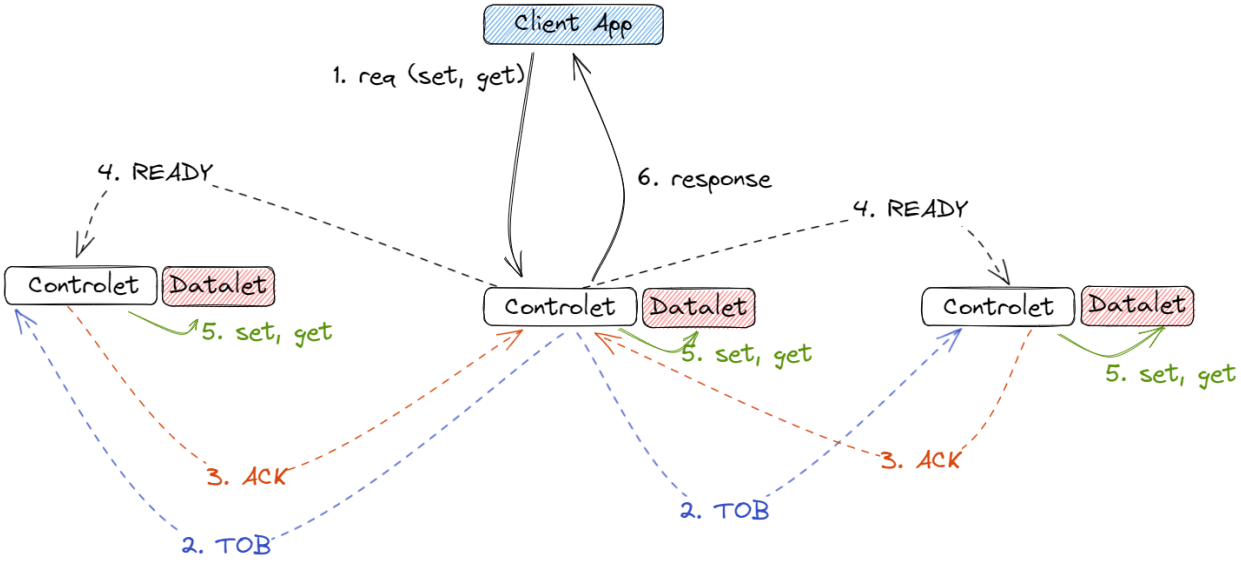
ARCHITECTURE

The architecture was influenced by the BESPO-KV paper. The system is divided into a Control plane and Data plane. The Control plane handles all the communications from the client, broadcasting, ordering from other replicas, and communication between its Datalet layer.

This separation creates nice hot swappable parts.
You can simply swap any off the shelf key value store with memcache support for the datalayer.

LINEAR CONSISTENCY:

DESIGN



IMPLEMENTATION

The architecture consists of two main components, the controlet and datalet. The controlet is responsible for managing the metadata for the system, such as the address of the datalets and the current state of the system. The datalet, on the other hand, is responsible for managing the actual key-value data and processing read and write operations. To ensure linear consistency, the system uses total order broadcast with Lamport clocks to order all operations in the system. Each operation is assigned a unique ID, which contains the Lamport clock value of the node that initiated the operation. The ID is included in the broadcast messages, which ensures that all nodes receive the same messages in the same order.

ALGORITHM:

A. When a request comes to the controlet, it increments its counter, generates a unique id for that operation.

B. The controlet now puts the request with its unique id in its local queue, with its lamport clock. The id is used to get to the status of the request in future operations. Then, the controller broadcasts the request to every other controlet with the unique id.
C. When a controlet receives a broadcast, the broadcast message consists of the unique id, the lamport clock, and then the request is added to its own local queue.
D. When a request is at the top of the queue, the acknowledgment is sent.
E. For a request that is at the top of the queue, and all the acknowledgements are received then the controlet broadcasts a “Ready” message to all the other servers.
F. When a “READY” msg is received for a request at the top of the queue, the request is popped from the queue and sent to the datalet.

Note: Whenever a request is added to the queue, the queue is sorted based on the lamport clocks.

This TOB is implemented for each operation such as GET/SET for linear consistency.

TESTING

The test cases are designed to match a real world scenario, the get and set are shuffled and then targeted at the system. This leads to for surety that the system can handle concurrent requests thrown at multiple servers as well, and should have linearizable history.

Run the file: “test_lc.py” for testing linearizability.
Test 1: Multiple SET requests
Test 2: Multiple GET requests
Test 3: Multiple GET, SET requests

In linear consistency, the write and read operations need to be ordered. Therefore, the system uses the total order broadcast 2 Ack method with Lamport clocks for ordering.
When a request is made, the GET/SET requests are broadcasted and sent to datelat in the same sequence.

The sequence of operations that each server performs is stored in the output/linear_consistency_output folder. Each file stores the order of requests it has processed, and that's how we test if every server is processing in the same order.
The system is tested against 3 replicas, a total of 15 get and set requests. The order of operation that resulted in shown below:

The format of the statement is
`reqid: {unique_request_id} {set/get} key {key}  {response from datalet}`

As you can see all the requests are processed in the same order across all the servers.

Server 0:

```
reqid: rBrwYWIJ set key iucxivy_1  True
reqid: sf4vnsz9 set key pkxmpsj_2  True
reqid: 08lH+bVW set key kddakwy_4  True
reqid: 6LSyx6Ij set key vlsvpdt_0  True
reqid: ct2U/ULV set key zthrbar_3  True
reqid: DDD64R/A get key kddakwy_4  b'hselojp'
reqid: Zhjmd06/ get key zthrbar_3  b'lirxpsi'
reqid: GUwXzzwd get key iucxivy_1  b'qngiyhx'
reqid: 1zkLjev6 get key vlsvpdt_0  b'sxkmulw'
reqid: G234Cwbs get key pkxmpsj_2  b'lqzbmaz'
reqid: dVtwhwLZ set key iucxivy_1  True
reqid: Wmgw8G/x get key zthrbar_3  b'lirxpsi'
reqid: JpMvxqbE set key zthrbar_3  True
reqid: pfuYzdoX get key pkxmpsj_2  b'lqzbmaz'
reqid: xFmrql5T get key zthrbar_3  b'tdcgyng'
file written to : 0
```

Server 1:

```
reqid: rBrwYWIJ set key iucxivy_1  True
reqid: sf4vnsz9 set key pkxmpsj_2  True
reqid: 08lH+bVW set key kddakwy_4  True
reqid: 6LSyx6Ij set key vlsvpdt_0  True
reqid: ct2U/ULV set key zthrbar_3  True
reqid: DDD64R/A get key kddakwy_4  b'hselojp'
reqid: Zhjmd06/ get key zthrbar_3  b'lirxpsi'
reqid: GUwXzzwd get key iucxivy_1  b'qngiyhx'
reqid: 1zkLjev6 get key vlsvpdt_0  b'sxkmulw'
reqid: G234Cwbs get key pkxmpsj_2  b'lqzbmaz'
reqid: dVtwhwLZ set key iucxivy_1  True
reqid: Wmgw8G/x get key zthrbar_3  b'lirxpsi'
reqid: JpMvxqbE set key zthrbar_3  True
reqid: pfuYzdoX get key pkxmpsj_2  b'lqzbmaz'
reqid: xFmrql5T get key zthrbar_3  b'tdcgyng'
file written to : 1
```

Server 2

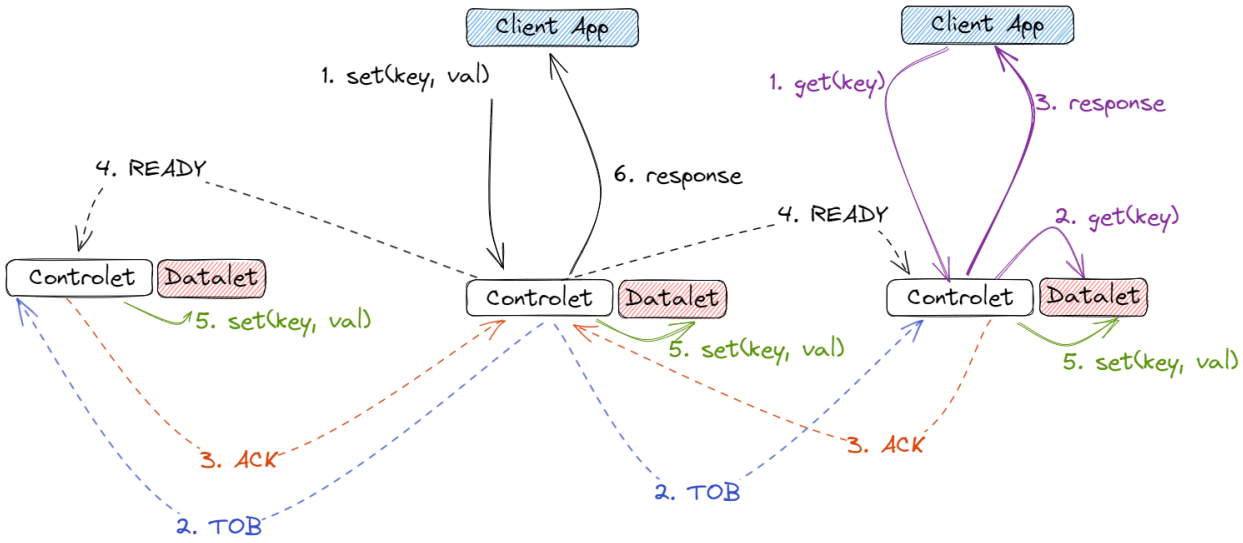
```
reqid: rBrwYWIJ set key iucxivy_1  True
reqid: sf4vnsz9 set key pkxmpsj_2  True
reqid: 08lH+bVW set key kddakwy_4  True
reqid: 6LSyx6Ij set key vlsvpdt_0  True
reqid: ct2U/ULV set key zthrbar_3  True
reqid: DDD64R/A get key kddakwy_4  b'hselojp'
reqid: Zhjmd06/ get key zthrbar_3  b'lirxpsi'
reqid: GUwXzzwd get key iucxivy_1  b'qngiyhx'
reqid: 1zkLjev6 get key vlsvpdt_0  b'sxkmulw'
reqid: G234Cwbs get key pkxmpsj_2  b'lqzbmaz'
reqid: dVtwhwLZ set key iucxivy_1  True
reqid: Wmgw8G/x get key zthrbar_3  b'lirxpsi'
reqid: JpMvxqbE set key zthrbar_3  True
reqid: pfuYzdoX get key pkxmpsj_2  b'lqzbmaz'
reqid: xFmrql5T get key zthrbar_3  b'tdcgyng'
file written to : 2
```

Linear Consistency test case output:

```
++++++++++ TEST 1: MULTIPLE SET OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
SET response  True
SET response  True
SET response  True
SET response  True
SET response  True
SET TEST CASE COMPLETED
++++++++++ TEST 2: MULTIPLE GET OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
GET response  b'hselojp'
GET response  b'lirxpsi'
GET response  b'qngiyhx'
GET response  b'sxkmulw'
GET response  b'lqzbmaz'
GET TEST CASE COMPLETED
++++++++++ TEST 3: MULTIPLE CONCURRENT SET AND GET  OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
SET response  True
GET response  b'lirxpsi'
SET response  True
GET response  b'lqzbmaz'
GET response  b'tdcgyng'
```

SEQUENTIAL CONSISTENCY:

DESIGN



IMPLEMENTATION:

To implement sequential consistency in the distributed key-value store, a combination of techniques was used. For GET operations, a local read protocol was implemented. This allowed nodes to read data locally without having to go through a consensus protocol, improving the performance and reducing the latency of read operations in the system. When a node receives a read request, it first checks its local copy of the data and returns it if it is up-to-date. Since the local read protocol guarantees that the data is up-to-date, there is no need to perform a consensus protocol for get operations.

For SET operations, Total Order Broadcast (TOB) was used to ensure that all nodes in the system receive messages in the same order. Each message is assigned a unique ID using a combination of the sender's Lamport clock value and a unique identifier for that node. When a node receives a message, it adds it to its incoming message queue and checks if all messages with smaller IDs have been delivered. If so, it adds the message to its deliverable queue and delivers all messages in the queue in the order of their IDs. This technique ensures that all nodes apply messages in the same order, providing linear consistency for the system. By using TOB for set operations, the performance of write operations is improved as they don't require waiting for consensus to be reached before committing the changes.

ALGORITHM

1. SET is the same as linear
2. GET is simple read from the datalet

TESTING

In sequential consistency, the reads are not ordered but the write needs to be ordered. So, the writes are TOB across the servers and they are sent to the datalets in a sequential order.
I have logged all the write operations, and tested if they are operated in the same sequences.

Run the test file: “test+sc.py” for sequential consistency.
Test 1: Multiple SET requests
Test 2: Multiple GET requests
Test 3: Multiple GET, SET requests

This was tested with 3 replicas and a total of 15 GET/SET requests in which 7/8 are SET requests.

This is order of all the write requests on all the servers:
Server 0:

```
reqid: nRWIxgQw set key xmtuciv_3  True
reqid: dg+214dW set key hoaaqij_4  True
reqid: L0vyz/Kt set key bigrbki_1  True
reqid: Lp7y//3Z set key nejivjb_2  True
reqid: FnpsysZh set key izroexq_0  True
reqid: 3A7lIAmm set key bigrbki_1  True
reqid: MJH69my2 set key xmtuciv_3  True
file written to : 0
```

Server 1:

```
reqid: nRWIxgQw set key xmtuciv_3  True
reqid: dg+214dW set key hoaaqij_4  True
reqid: L0vyz/Kt set key bigrbki_1  True
reqid: Lp7y//3Z set key nejivjb_2  True
reqid: FnpsysZh set key izroexq_0  True
reqid: 3A7lIAmm set key bigrbki_1  True
reqid: MJH69my2 set key xmtuciv_3  True
file written to : 1
```

Server 2:

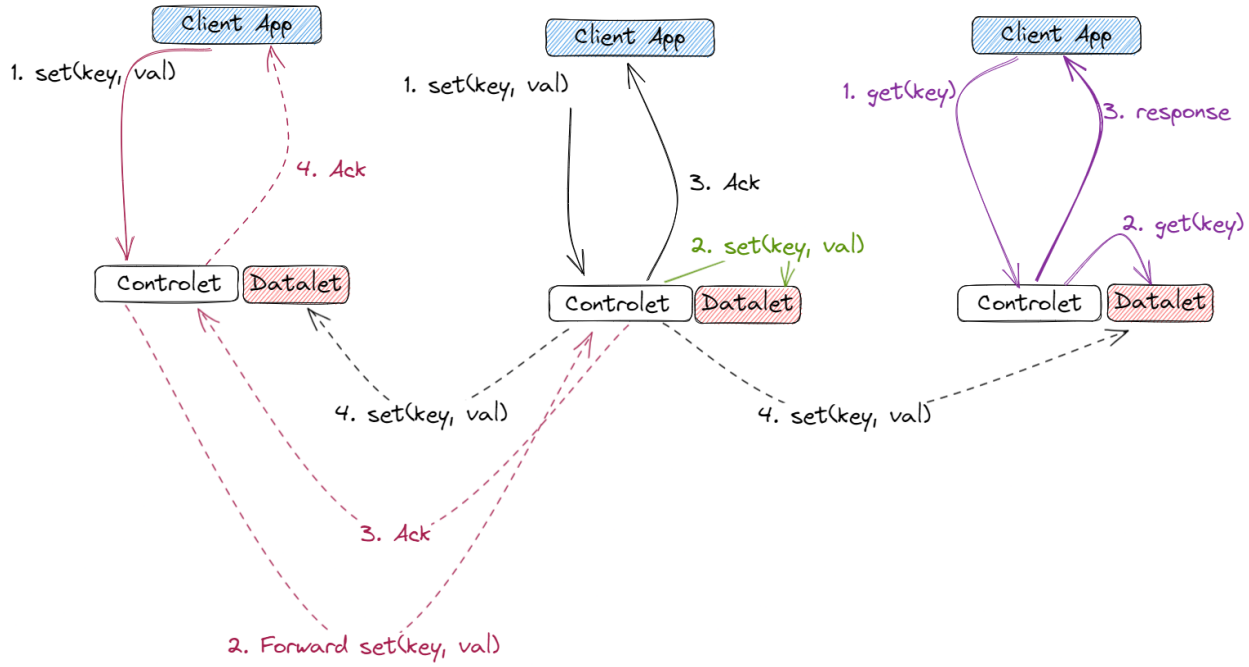
```
reqid: nRWIxgQw set key xmtuciv_3  True
reqid: dg+214dW set key hoaaqij_4  True
reqid: L0vyz/Kt set key bigrbki_1  True
reqid: Lp7y//3Z set key nejivjb_2  True
reqid: FnpsysZh set key izroexq_0  True
reqid: 3A7lIAmm set key bigrbki_1  True
reqid: MJH69my2 set key xmtuciv_3  True
file written to : 2
```

Sequential Consistency Test cases output.

```
FINISHED CONTROL PLANE AND DATA PLANE INITIALIZATION
++++++++++ TEST 1: MULTIPLE SET OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
SET response  True
SET response  True
SET response  True
SET response  True
SET response  True
SET TEST CASE COMPLETED
++++++++++ TEST 2: MULTIPLE GET OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
GET response  b'yahjtao'
GET response  b'rtgfgoa'
GET response  b'krthbjf'
GET response  b'lbydztc'
GET response  b'jwktgwm'
GET TEST CASE COMPLETED
++++++++++ TEST 3: MULTIPLE CONCURRENT SET AND GET  OPERATIONS ON MULTIPLE SERVERS++++++++++++++++++
GET response  b'krthbjf'
GET response  b'krthbjf'
GET response  b'rtgfgoa'
SET response  True
SET response  True
test completed 5
```

EVENTUAL CONSISTENCY:

DESIGN



IMPLEMENTATION

To achieve eventual consistency in the distributed key-value store, a local write protocol was implemented. This protocol is used to ensure that all write operations performed on the system are eventually propagated to all datalets, thereby ensuring that all clients have consistent access to the most up-to-date version of the data.

In this protocol, each key is assigned to one of the controlets, which are responsible for all write operations related to that key. The controlets maintain a counter for each key that is updated for every write operation. This counter ensures that there are no race conditions when multiple write operations are performed concurrently.

When a client sends a write request for a key to a specific controlet, that controlet first calculates the hash of the key. The hash function is designed in a way that ensures that the hash value for each key is unique and evenly distributed across all available controlets. Based on the hash value, the controlet then determines which datalet is responsible for storing the data for that key.

If the client sends a request for a key to a controlet that is not responsible for that key, the controlet forwards the request to the correct controlet. This forwarding process ensures that the write request is always directed to the correct controlet responsible for the data for that key.

When a write request is made to a controlet, it makes sure it is responbsile for that key otherwise forwards to the correct controlet.

The responsible controlet then forwards the request to its respective datalet. The datalet first checks if the incoming flag (which corresponds to the counter of the request) is higher than the current flag for that key. If the incoming flag is higher, it means that the incoming request is a more recent update than the current version, and the datalet updates its local copy of the data.

Once the datalet has updated its local copy of the data, the respective controlet sends acknowledgement to the client and then broadcasts the write request to all other datalets in the system with its own counter as flag. The same comparison logic is applied at every datalet. This ensures that the most recent update is propagated to all datalets and that eventually all datalets will have the most up-to-date version of the data. This leads to faster responses to clients and eventual consistencies.
The local write protocol ensures eventual consistency in the system, as there may be a delay in propagating updates to all datalets in the system, resulting in temporary inconsistencies. However, the protocol guarantees that all updates will eventually be propagated to all datalets, thereby ensuring that all clients have consistent access to the most up-to-date version of the data.

For read operation the controlet, just sends the request directly to the datalet and the response back to the client directly.

ALGORITHM

1. SET is responded directly back to the client and has an async broadcast,
2. GET is simple read from the datalet

TESTING

Exhibiting stale reads, the test case is in the test_ec.py file.

```
++++++++++++++ TEST 1: SHOWCASING STALE READ  ++++++++++++++
Setting the initial value of the key 'test_key' to 1
SET response  True
Checking the value is updated everywhere
GET response  b'1'
GET response  b'1'
GET response  b'1'
GET response  b'1'
GET response  b'1'
Now starting a new write to set 'test_key' to 2
time.sleep introduces delay in the broadcast.
Showcasing the stale read, After a write operation it should be have been 2 in a linear mannaer, but it is 1.
SET response  True
GET response  b'1'
Waiting for the write to complete
++++++++++++++ TEST 2: SHOWCASING EVENTUAL WRITE  ++++++++++++++
Showcasing the updated value from the write should be 2 after enough time has passed.
GET response  b'2'
GET response  b'2'
GET response  b'2'
GET response  b'2'
GET response  b'2'
```

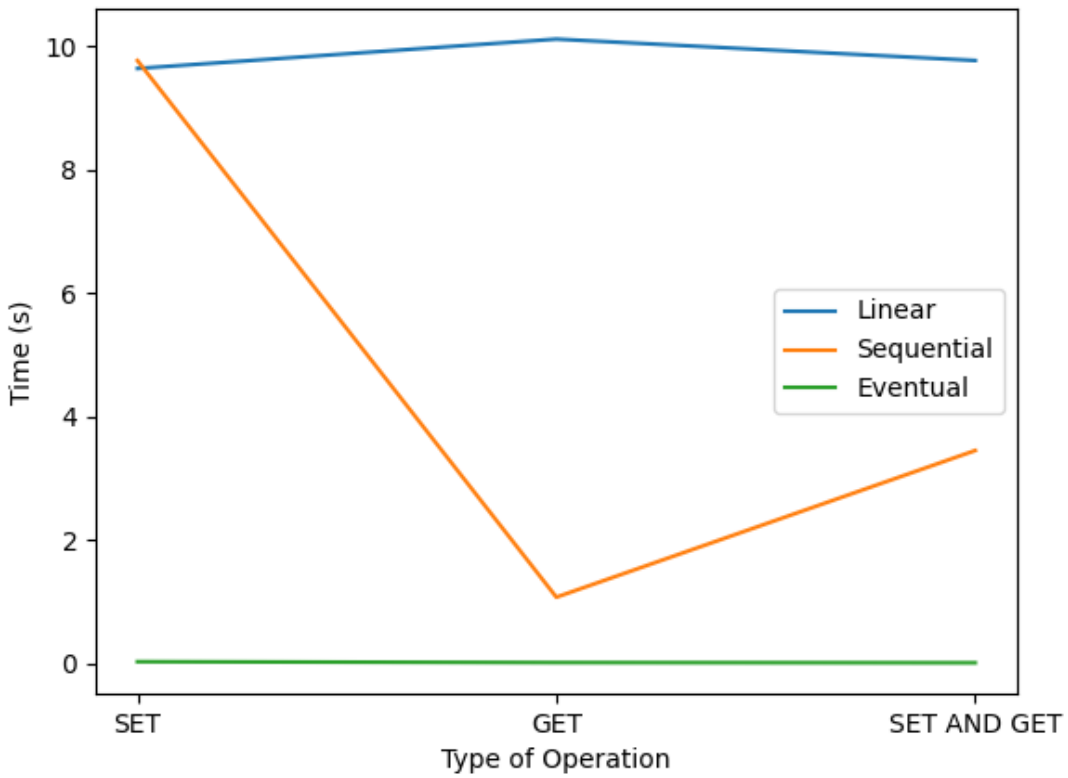
MAIN CHALLENGES

1. Deciding on the architecture for the consistencies.
2. Deciding on the implementation of each consistency.
3. Handling the race conditions while implementing the Total Order Broadcast.

PERFORMANCE TESTING:

The system was set to three replicas and is tested against 100 SET requests at a time, 100 GET requests at a time and 100 SET, GET requests at a time.

	SET requests	GET requests	SET and GET requests
Linear Consistency	9.6444002866745	10.120154695510	9.773513970375062
Sequential Consistency	9.771539509296417	1.0758793091773	3.45280016660690
Eventual Consistency	0.030848305225372	0.0165560150146	0.012655572891235



CONSTRAINTS, LIMITATIONS AND ASSUMPTIONS

1. Number of servers is always greater than or equal to 3.
2. If the number of requests is too high, there are unexpected results.
3. The value cannot be longer than 4000 bytes.

REFERENCES

1. BESPOKV
2. Slides from the professor