Assignment-4:
Distributed Multi-Consistency Key-Value Store

Submitted by: Priti Singh

Introduction

For this programming assignment, my task was to design and implement a distributed multi consistency key value store which shows the different consistency schemes that use the underlying key-value store replicas. I have used the key vaue store from the first assignment which takes the input in the following manner

Set

The set command is whitespace delimited, and consists of **two** lines:
set <key> <value> \r\n
The server should respond with either "STORED\r\n", or "NOT-STORED\r\n".

Get

Retrieving data is simpler: get <key>\r\n

The server should respond with two lines:
VALUE <key> <bytes> \r\n
<data block>\r\n

I have spawned multiple processes for multiple servers that could handle multiple clients.

The following are the architecture , design and test cases for each of the model.

Architecture – Eventual Consistency

I have implemented the broadcasting method to achieve this. I have spawned multiple servers for this and these servers can be connected to multiple clients. The server which gets connected to client writes to its local copy first and then broadcast it to other servers which keep listening.

Once they receive the broadcast message, they update their key value store.

For read the client can be connected to any server and there are also chances that it may receive stale reads
On a high level, the following steps were taken to achieve eventual consistency

For servers:

1. Each server maintains a copy of the data.
2. When a client sends a write(set) request, the server updates its own copy of the data.
3. The server then propagates the update to all other servers in the system.
4. The other servers eventually receive the update and apply it to their own copies of the data.

For clients:

1. Each client can read from any server in the system.
2. When a client sends a read request, it may receive different responses from different servers.
3. If the client sends a write request, it sends the request to any server in the system.

4. The server will update its own copy of the data and eventually propagate the update to all other servers in the system.

Design

1: Server

The servers are processes in the system that receive the client request and the broadcasted message from the other servers who has responded to client.
I have spawned multiple servers using processes. As soon as the client is served, it starts a broadcast to other servers. I have used zeroMQ PUB-SUB architecture for this .The server which is connected to client becomes the publisher and broadcast the message to subscribers(other servers)

3: Client

The client is a process that wants to send a request set/get to all servers in the system. The client choses the server randomly

Code Structure

ServerEventual.py and ClientEventual.py should be run in different terminal.

Test Cases:

I have spawned 5 servers and 2 clients are connected to them at the same time for the below screenshots.

client.py :

```
r DistributedKeystore % python ClientEventual.p
y
Client connected to server 6053 T 04:21:42
Enter command: set finaloo 8
Received from server: b'STORED' T 04:22:03
(base) priti@Pritis-MacBook-Air DistributedKeys
tore %
```

```
DistributedKeystore % python clientEvent
ual.py
Client connected to server 6050 T 04:21:
55
Enter command: get finaloo
Received from server: KEY NOT FOUND T 04
:22:04
(base) priti@Pritis-MacBook-Air Distribu
tedKeystore %
```

Server.py

```
(base) priti@Pritis-MacBook-Air DistributedKeystore % python ServerEve
ntual.py
Listening ...
Server started listening on 6050 T 04:21:40
Server started listening on 6051 T 04:21:40
Server started listening on 6052 T 04:21:40
Server started listening on 6053 T 04:21:40
Server started listening on 6054 T 04:21:40
Server on 6053 responded back to client T 04:22:03
Server 6053 stated broadcasting T 04:22:03
Server 6050 received broadcasted message 'set finaloo 8'  04:22:03
Server on 6050 responded back to client T 04:22:04
Server 6050 stated broadcasting T 04:22:04
Server 6051 received broadcasted message 'get finaloo'  04:22:04
Server 6052 received broadcasted message 'get finaloo'  04:22:04
Server 6053 received broadcasted message 'get finaloo'  04:22:04
Server 6054 received broadcasted message 'get finaloo'  04:22:04
Server 6050 updated key:finaloo value:8 T 04:22:24
Server 6051 received broadcasted message 'set finaloo 8'  04:22:24
Server 6051 updated key:finaloo value:8 T 04:22:44
Server 6052 received broadcasted message 'set finaloo 8'  04:22:44
Server 6052 updated key:finaloo value:8 T 04:23:04
Server 6054 received broadcasted message 'set finaloo 8'  04:23:04
Server 6054 updated key:finaloo value:8 T 04:23:24
```

The above test case demonstrates that when a write(set) is performed on server 6053 to write the key "finaloo" and if before broadcasting this write a another request is given to server 6050 it will return "Key not found" response because the message has not been received by server 6050
but if after sometime reads the value it will get the updated value.

clientEventual.py

```
edKeystore % python clientEventual.py
Client connected to server 6050
Enter command: get finaloo
Received from server: VALUE finaloo 1
8
```

Architecture – Sequential Consistency:

I have implemented sequential consistency using primary based model. The write request (set) is first sent to primary and primary updates its key value store first and broadcast the message to other severs replicas after receiving it sends an acknowledgement to the primary and after this replica sends the response to the client.

Server Algorithm:

1. Listen for incoming requests from the clients.
2. If the request is a write request it sends to the primary, after receiving the broadcast message it update the key-value store with the new value and respond to client.
3. If the request is a read request, return the current value from the key-value store.

Primary Server Algorithm:

1. Listen for incoming write requests from the client as well as replicas
2. When a write request is received, update the key-value store with the new value
3. Broadcast the write request to all replicas
4. Wait for acknowledgement from all replicas

Design

1: Server Replica

The servers are processes in the system that receive the client request. If it is a write request is made by the client it sends the request to the primary server through zeroMQ REP-REQ(response request socket).Each server is also a subscriber port which listens to the broadcast message form the primary server.
After receiving the broadcast message each server updates its key value store to maintain a consistent view of the system.

Primary Server:

It is a designated server. It can also serve the client as well , if the client connects to it ,it will be broadcasting and waiting for the acknowledgement and then responding to the client.
For usual cases when it is not connected to client it will keep listening to the write message that's been forwarded by replica and proceed further according to the algorithm

3: Client

The client is a process that wants to send a request set/get to all servers in the system. The client choses the server randomly including primary server.

Code Structure

Primary.py, ServerSeq.py and ClientSeq.py should be run in different terminal.

Test Cases:

Test case 1:Set Request
I have spawned 5 servers and 2 clients are connected to them at the same time for the below screenshots.

ServerSeq.py

```
Sending to primary set test5 5
Server on 5052 responded back to client T 17:14:43
Server 5051 replica received message from primary
Server 5052 replica received message from primary
Server 5050 replica received message from primary
Server 5054 replica received message from primary
Server 5053 replica received message from primary
Server 5054 updated
Server 5051 updated
Server 5053 updated
Server 5050 updated
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5054 replica sends the ack
Server 5050 replica sends the ack
Server 5054 replica received message from primary
Server 5050 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5054 replica sends the ack
Server 5050 replica sends the ack
Server 5054 replica received message from primary
Server 5050 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5054 replica sends the ack
```

Primary.py

```
Primary server received from replica set test5 5
STORED
Primary Server 8873 stated broadcasting
Primary server received ack from replica 4071
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4072
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4071
Primary server received ack from replica 4074
Primary server received ack from replica 4073
Primary server received ack from replica 4070
Primary server received ack from replica 4072
Primary server received ack from replica 4071
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4073
Primary server received ack from replica 4071
Primary server received ack from replica 4073
Primary server received ack from replica 4074
Primary server received ack from replica 4072
Primary server received ack from replica 4070
Primary server received ack from replica 4071
```

Client.py

```
eystore % python clientSeq.py
Client connected to server 5052 T 17:14:35
Enter command: set test5 5
Received from server: b'STORED' T 17:14:43
(base) priti@Pritis-MacBook-Air DistributedK
eystore %
```

This test case demonstrates the writing of key test5 to value 5. First it sends the request to primary as shown in screenshot “Primary.py” and broadcast the message to other servers which listens and sends the ack after updating to each of the key value store.

data > key_value_store_8873.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

data > key_value_store_5054.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

data > key_value_store_5052.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

data > key_value_store_5053.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

data > key_value_store_5050.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

data > key_value_store_5051.txt

```
1 test:1
2 test3:3
3 test4:4
4 test5:5
5
```

<u>Test case2: Set request</u>

To demonstrate what happens when value is already present following is the test case , it should return NOT STORED to the client.

ServerSeq.py.

```
python serverSeq.py
Listening ...
Server started listening on 5050
Server started listening on 5051
Server started listening on 5054
Server started listening on 5052
Server started listening on 5053
Server on 5054 responded back to client T 17:17:23
Sending to primary set test4 4
Server on 5051 responded back to client T 17:18:29
Server 5051 replica received message from primary
Server 5050 replica received message from primary
Server 5052 replica received message from primary
Server 5053 replica received message from primary
Server 5054 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5052 replica sends the ack
Server 5054 replica sends the ack
Server 5052 replica received message from primary
Server 5054 replica received message from primary
Server 5050 replica sends the ack
Server 5053 replica sends the ack
Server 5050 replica received message from primary
Server 5053 replica received message from primary
Server 5052 replica sends the ack
Server 5053 replica sends the ack
Server 5052 replica received message from primary
Server 5053 replica received message from primary
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5054 replica sends the ack
Server 5051 replica sends the ack
Server 5054 replica received message from primary
Server 5051 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5054 replica sends the ack
Server 5052 replica sends the ack
Server 5053 replica sends the ack
Server 5054 replica received message from primary
Server 5053 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
```

Primary.py

```
python Primary.py
Listening ...
Primary Server started listening on 8873
Primary server received from replica set test4 4
NOT_STORED
Primary Server 8873 stated broadcasting
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4073
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4072
Primary server received ack from replica 4070
Primary server received ack from replica 4073
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4072
Primary server received ack from replica 4074
Primary server received ack from replica 4073
Primary server received ack from replica 4071
Primary server received ack from replica 4070
Primary server received ack from replica 4072
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4070
```

ClientSeq.py

```
eystore % python clientSeq.py
Client connected to server 5051 T 17:18:04
Enter command: set test4 4
Received from server: b'NOT STORED' T 17:18:
29
```

Test case 3: Get (Read) request

The Get in Sequential will not be forwarded to the primary instead it will be served from the local copy .
The following “Primary.py” demonstrates that no request was sent to it from the replica when client issued a get request.

ServerSeq.py

```
python serverSeq.py
Listening ...
Server started listening on 5050
Server started listening on 5051
Server started listening on 5054
Server started listening on 5052
Server started listening on 5053
Server on 5054 responded back to client T 17:17:23
```

Primary.py

```
python Primary.py
Listening ...
Primary Server started listening on 8873
```

ClientSeq.py

```
eystore % python clientSeq.py
Client connected to server 5054 T 17:17:17
Enter command: get test4
Received from server: VALUE test4 1
4

 T 17:17:23
```

Architecture- Linearizability

I have implemented linearizability using the primary based protocol itself. Since it is a stricter the only difference is the reads are also blocking it this. The reads(get) request is also sent to primary first and only after the getting the appropriate value from the primary the replica responds to client. This will make sure that client gets a coherent view all the time.

Server Algorithm:

1. Listen for incoming requests from the clients.
2. The request are sent to the primary, after receiving the broadcast message it update/read the key-value store with the new value and respond to client.

Primary Server Algorithm:

1. Listen for incoming requests from the client as well as replicas
2. When request is received, update the key-value store with the new value or reads the value
3. Broadcast the request to all replicas
4. Wait for acknowledgement from all replicas

Design:

1: Server Replica

The servers are processes in the system that receive the client request. If it is a write or read request is made by the client it sends the request to the primary server through zeroMQ REP-REQ(response request socket).Each server is also a subscriber port which listens to the broadcast message form the primary server. After receiving the broadcast message each server updates its key value store to maintain a consistent view of the system.

2: Primary Server:

It is a designated server. It can also serve the client as well , if the client connects to it ,it will be broadcasting and waiting for the acknowledgement and then responding to the client.
For usual cases when it is not connected to client it will keep listening to the message both read and write that's been forwarded by replica and proceed further according to the algorithm

3: Client

The client is a process that wants to send a request set/get to all servers in the system. The client choses the server randomly including primary server.

Test Case:

I have spawned 5 servers , when a client sends get request , it is given it primary first as shown in the following screenshots.

ClientLin.py

```
(base) priti@Pritis-MacBook-Air DSFinal % python cl
ientLin.py
Client connected to server 5052 T 20:08:40
Enter command: get test4
Received from server: KEY NOT FOUND T 20:09:07
(base) priti@Pritis-MacBook-Air DSFinal %
```

ServerLin.py

```
(base) priti@Pritis-MacBook-Air DSFinal % python ServerLin.py
Listening ...
Server started listening on 5050
Server started listening on 5052
Server started listening on 5051
Server started listening on 5053
Server started listening on 5054
Sending to primary get test4
Server 5052 replica received message from primary
Server 5051 replica received message from primary
Server 5054 replica received message from primary
Server 5053 replica received message from primary
Server 5050 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5054 replica sends the ack
Server 5053 replica sends the ack
Server 5054 replica received message from primary
Server 5053 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5052 replica sends the ack
Server 5052 replica received message from primary
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5052 replica sends the ack
Server 5054 replica sends the ack
Server 5054 replica received message from primary
Server 5050 replica sends the ack
Server 5050 replica received message from primary
Server 5050 replica sends the ack
Server 5054 replica sends the ack
Server 5051 replica sends the ack
Server 5051 replica received message from primary
Server 5053 replica sends the ack
Server 5053 replica received message from primary
Server 5053 replica sends the ack
Server 5051 replica sends the ack
```

PrimaryLin.py

```
(base) priti@Pritis-MacBook-Air DSFinal % python Prim
aryLin.py
Listening ...
Primary Server started listening on 8873
Primary server received from replica get test4
Primary Server 8873 stated broadcasting
Primary server received ack from replica 4072
Primary server received ack from replica 4073
Primary server received ack from replica 4074
Primary server received ack from replica 4072
Primary server received ack from replica 4071
Primary server received ack from replica 4070
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4072
Primary server received ack from replica 4074
Primary server received ack from replica 4070
Primary server received ack from replica 4071
Primary server received ack from replica 4073
Primary server received ack from replica 4072
Primary server received ack from replica 4074
Primary server received ack from replica 4070
Primary server received ack from replica 4070
Primary server received ack from replica 4074
Primary server received ack from replica 4071
Primary server received ack from replica 4073
Primary server received ack from replica 4073
Primary server received ack from replica 4071
```

The writing is similar to Sequential consistency.

Architecture - Causal Consistency

To achieve causal consistency, it must be ensured that the order of operations (updates and reads) is consistent with the causal dependencies between them. I have implemented this using vector clocks to track the causal dependencies between operations.

Test Case:

There are 4 servers and 3 clients , so the client should always be able to read the latest value and if multiple writes are performed. For this case test3 latest value was 3 which was send to client.

ClientCau.py 1

```
(base) priti@Pritis-MacBook-Air DSFinal % python clientCau.py
Client connected to server 5046
Enter command: get test3
Received from server: VALUE test3 1
4
```

ClientCau.py 2

```
(base) priti@Pritis-MacBook-Air DSFinal % python clientCau.py
Client connected to server 5048
Enter command: set test3 4
Received from server: b'STORED'
```

ClientCau.py 3

```
(base) priti@Pritis-MacBook-Air DSFinal % python clientCau.py
Client connected to server 5045
Enter command: set test3 3
Received from server: b'STORED'
```

ServerCau.py

```
(base) priti@Pritis-MacBook-Air DSFinal % python ServerCau.py
Listening ...
Server started listening on 5045
Server started listening on 5046
Server started listening on 5047
Server started listening on 5048
Server 0 broadcasted message '2,0,0,0 set test3 3'
Server 5046 received broadcasted message '2,0,0,0 set test3 3'
STORED
Server 5047 received broadcasted message '2,0,0,0 set test3 3'
STORED
Server 5048 received broadcasted message '2,0,0,0 set test3 3'
STORED
Server 3 broadcasted message '0,0,0,2 set test3 4'
Server 5045 received broadcasted message '0,0,0,2 set test3 4'
STORED
Server 5046 received broadcasted message '0,0,0,2 set test3 4'
STORED
Server 5047 received broadcasted message '0,0,0,2 set test3 4'
STORED
Server 1 broadcasted message '0,1,0,0 get test3'
Server 5045 received broadcasted message '0,1,0,0 get test3'
VALUE test3 1
4
END

Server 5047 received broadcasted message '0,1,0,0 get test3'
VALUE test3 1
4
END

Server 5048 received broadcasted message '0,1,0,0 get test3'
VALUE test3 1
4
END
```

The data store contains multiple entries for a single value but the max vector is always returned to client.

data > key_value_store_5048.txt

```
1    test1:2:3,0,0,0
2    test1:2:3,3,0,0
3    test3:3:3,0,0,0
4    test3:4:3,0,0,1
5
```

Assumptions

1. I have assumed that there is no network failure while data is transferred between each component over socket.
2. I have also assumed that the Primary server node does not fail for the sequential and linearizability implementation.

Limitations

For the sequential and linearizability implementation:
All the load on Primary: sequencer being a bottleneck and the risk of the entire algorithm failing if the primary process fails.
For causal , multiple entries are stored which may result in taking huge space.This can be mitigated by keeping only one value.