Assignment 4- Report

Distributed KV Store

- Mettukuru, Avinash Reddy

Overview

This assignment is an implementation of Distributed Key-Value stores with different consistency models such as eventual, sequential, and linearizable.

Design

- In this implementation to make the coding part easy and not to have different , I have assumed that UDP is reliable and follows FIFO.
- In each partition there is a multicast socket listening for all the broadcasts made from all the different partitions running. Unfortunately, Luddy server's didn't provide support multicast group's, after contacting *Sadiwala, Adit Rajesh* and guided me to test it on WSL, the implementation worked perfectly in **WSL**, so please ***run this implementation locally with JDK8*** or above.
- I have implemented an Abstract template for partition, that contains core features as creating and joining the multicast group, connection to KV server, abstracts SET GET functions.
- For both sequential, linearizability consistency implementations I have used HashMap that counts the acknowledgments received for the total order multicast algorithm and also client socket address.
- For Eventual, we make use of multicast socket to send the information about recent writes to all partitions.

Implementation

Multicast Socket

Each partition has a multicast socket at "234.0.0.0:4446" listening for all the message broadcasts made by other partitions for total order multicast algorithm(make sure to run on systems with multicast support).

Priority Queue

I have used *Java's PriorityQueue<T> with Message* object which contains timestamp, sender's process Id, message Id and sender InetAddress. To compare(order) message objects I have used timestamp's and to break ties process id is used.

Total Order Multicast

Each partition maintains a HashMap which counts the number of acknowledgments received for a message referred to by message id, generated using UUID module in java. Whenever a message is at top and acknowledgement is not sent, we will send the acknowledgement and set the flag in message as true if the message is its own or remove the message from queue. When a partition receives an acknowledgement, it updates the acknowledgement count in the HashMap, and when the count reaches the total number of partitions the operation is then executed.

Build & Run

Project can be built and tested on Linux machines with JDK8 or above. All additional libraries included in submitted zip file.

How to run

- Open terminal and navigate to **"KVstore"** and using the following make command "***make runKvServer port=8080 cache_path=cache.json***" to start our Memcached kv store for the partitions to store the Key-Value pairs.
- In a separate terminal navigate to **"Linearizability"** and use the following command "***make runLinear partitions=3***" to start the partitions that implements Linearizable consistency. The "**partitions"** CL argument denotes the number of partitions needs to be started. After all the partitions have started successfully the ports for each partition is logged into the console.
- To start partitions that implement sequential consistency navigate to "**Sequential**" directory and use the following command "***make runSequential partitions=3***" to start the partition, this follows the same partitions CL argument notation and also logging port numbers of partitions as in linearizability implementation.
- In similar to above consistency models, navigate to "**Eventual**" directory and use the commend "**make runEventual partitions=3**" to start 3 partitions.
- To start our client open as many separate terminal as many clients necessary and navigate to "**src"** directory and run the client using the following command "***make runClient***" and follow the on display instructions to proceed forward.

Tests

- For edge cases where a non-exist key is accessed, partitions return "Error" as a response.

Linearizability

- Using the text-book examples I have tested the correctness of the algorithm, where two client, say P and Q, and three partitions, say 1,2,3, the execution history follows:
    - Q.set(x,1) to 1:P.get(x)to 2:Q.ok()from 1:P.ok(1)from 2:Q.get(x) to 3:Q.ok(1) from 3
- To make the process Q delayed response I have used Thread.sleep().

Sequential

- In like to test case used to test linearizability, I have used the exact same configuration and the execution history follows:
    - Q.set(x,1) to 1:P.get(x)to 2:P.ok(Error)from 2:Q.ok()from 1:Q.get(x) to 3:Q.ok(1) from 3
- We can observe that the client P received a response corresponds to stale key read for read(x), as the write to x is delayed, this is due to local read algorithm used in sequential consistency we got an instant response.

Eventual

- Similar to Linearizability I have used the exact same configuration and the execution history follows:
    - Q.set(x,1) to 1:Q.ok() from 1:P.get(x) to 2:P.Ok(Error) from 2:P.get(x) to 2:P.Ok(1) from 2

Performance

- Each partition spawns three other threads to listen for broadcast, client requests, total order multicast algorithm(only in sequential, linearizability models), which can be very hungry on the CPU clocks.

Limitations

- To receive and store UDP packets I have used 256byte long arrays, so that applies a restriction on any message that contains key-value pairs longer than 256byte the end gets cut-off.
- Also UDP is not reliable in production grade servers, any lost message or acknowledgment might cause

Improvements

- Instead of using UDP packets we can use one connection to a message broker such as RabbitMQ, Apache Kafka, Java Messaging Service(JMS) to make the networking reliable and actually follow FIFO ordering of messages.
- We could also implement logical clocks for a better sound total ordering multicast algorithm.