Distributed Key Value store

Submitted by: Sumant Gaopande

For all my implementations of different consistency models I have used my **own key-value store** to focus more on the correctness of my models. All my key-value stores are **multi-threaded** servers that can accept concurrent SET/ GET requests.

My submission is arranged as 3 folders each containing the code for Sequential consistency, Linearizability & Eventual consistency respectively.

To run my programs please install Python3 and have Python3 to be the default interpreter. i.e when python 'xyz.py' is run it should pick up python3 as the default interpreter. If it's not you can do that with:

$ echo "alias python='python3'" >> .bashrc

$ source .bashrc

Each implementation has a **'driver.py'** file that defines various test parameters which is basically the driver process that spawns multiple key-value store & client processes. This has been done to simplify testing and execution. Each implementation comes with a **'test_case.json'** file that describes various test cases for each consistency model. You can also define your own case. Please see the **main** part of 'driver.py'.

For any test case you must define 4 variables:

- **num_processes** = 3, which defines how many key-value processes you want to spawn.
- **ports** = [8081, 8082, 8083], which defines a list of ports you want for the key-value store processes.
- **output_sleep** = 7, A random amount of time that you think would be needed for the communication of all messages, after which a request can be made to the key-value store processes to display the store data & final order of the messages in terms of

Sequential consistency and **Linearizability**. This sleep has been incorporated in 'driver.py' to make it easier for the user to view the result.

The 'test_case.json' file contains various test cases where the format is:

```
"test_case_number": {
        "client_number": [
          {"message": {"type": "GET", "key": "m1", "delay": 0}, "port": 8082},
          {"message": {"type": "STORE", "key": "m1", value: "m1", "delay": 0}, "port": 8082}
        ]

    }
```

Sequential Consistency:

Implementation

My implementation is based on the **Local Read protocol** with the strong assumption that the total ordered multicast works without failures**.** I have not deviated from this algorithm and my implementation reflects a simple design to implement it.

For a read:

Whenever a client requests a read to a replica - a local read result is delivered to it i.e. a value is returned to it based on the local replicas data store.

For a write:

Whenever a client requests a write to a replica - A totally ordered multicast is initiated from the replica the client has requested a write to. Each replica maintains a Queue sorted by a logical Lamport timestamp. Once a write is at the front of the queue and all acknowledgements have been received and the key value pair is committed to the store &

popped from the queue, does the initiating process reply back to the client indicating a successful write.

Details:

- For my Totally ordered multicast queue I have used a **minimum heap.** Each entry in the queue has a tuple whose first two values are - (Lamport timestamp, Process id). This ensures that the lowest Lamport timestamp & process id combination is popped from the queue. One **advantage of using a minimum heap** over a normal list that is sorted, is that it reduces the time-complexity of ordering the queue every time an entry is added to the queue from **O(nlogn) to O(logn)**.
- The writes are blocking as it should be according to the local read algorithm.
- All communication is done via TCP sockets.

Challenges

The biggest challenge has been testing the model. To simplify it I have kept a list called 'final' that appends the order in which the writes are committed by all the replicas. Comparing them I can prove that my implementation works and that the replicas display a consistent order of operations as required by Sequential consistency even when random delays are introduced with the requests. Other than that, my log files for each process display all the details required to view the order of operations by the clients & the replicas.

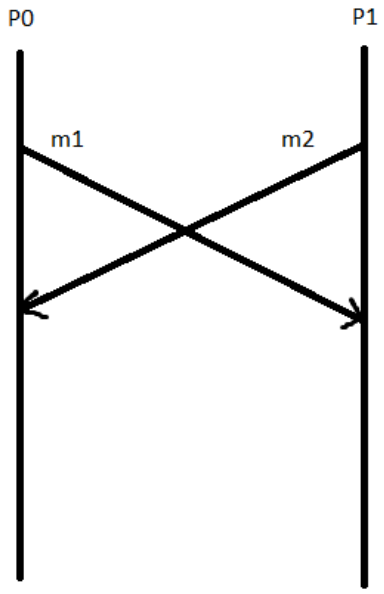
Performance evaluation

In terms of performance, as discussed in most of the lectures describing strict consistency models - this implementation is a slow implementation for a write heavy system and would be difficult to manage in a production system that needs to handle crashes and failures. As writes are blocking, a client only receives a response when a totally ordered broadcast is finished for that write which is a slow response time. Reads are fairly easy and fast.

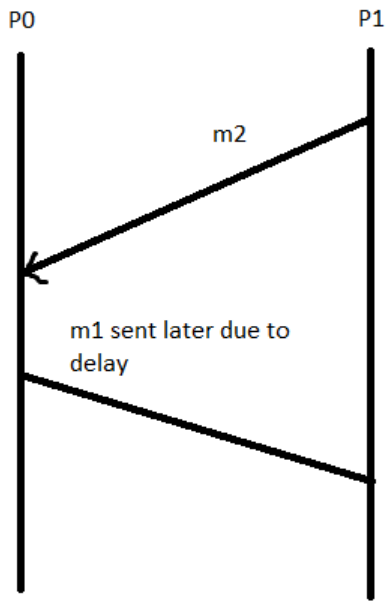
Testing

Please give the test cases ample time to run - one of the test cases takes nearly 30 seconds to complete a run to simulate random delays that have been put in to imitate real world randomness in the network.

The below 2 test cases prove the correctness of the program for the most trivial case for Totally ordered messages.



- This is the case 1 described in my program found in the main part of driver.py.
- The order that is observed is message m1 is committed first & m2 later sent by concurrent clients.



- This is the case 2 described in my program found in the main part of driver.py.
- Although m1 & m2 were sent to both the processes nearly at the same time adding a delay to m1 caused m2 to be delivered first & getting a lower logical timestamp compared to m1.
- The order that is observed by the processes is m2 being committed first then m1.

You can also view & run various test cases defined in the 'test_case.json' file.

Linearizability:

Implementation

My implementation is based on the modified **Local Read protocol** that was discussed in class**.** Again to clarify, I have not deviated from this algorithm and my implementation reflects a simple design to implement it. The simple idea is to issue a totally ordered multicast for **both reads & writes** as opposed to only writes for Sequential consistency. In a broad sense it looks as if all the replicas execute the Get & set operations in the same order appearing as if the user is interacting with a single store.

For a write:

Similar to the writes for a Local read algorithm for Sequential consistency whenever a client requests a write to a replica - A totally ordered multicast is initiated from the replica the client has requested a write to and that request is pushed in a queue. Each replica maintains a queue sorted by a logical Lamport timestamp. Once a write is at the front of the queue and all acknowledgements have been received and the entry is popped from the queue and a key value pair is committed to the store does the initiating process reply back to the client indicating a successful write.

For a read:

Now modifying the Local read algorithm to have reads to be totally ordered & multicast - whenever a client requests a read to a replica - that read is pushed in a queue & a totally ordered multicast is initiated. Each replica maintains a queue sorted by a logical Lamport timestamp. Once a read is at the front of the queue and all acknowledgements have been received for the broadcast of the read & the entry is popped from the queue does the initiating process reply back to the client indicating a read value by reading from the store at that moment.

Challenges

Similar to Sequential consistency the biggest challenge has been testing the model - proving a total order of read & write messages. In addition to the write messages being appended in the ‘final’ queue, the ‘reads’ are also logged in the ‘final’ queue that can be compared across the replicas to get a consistent total order of both read 7 write messages for all the replicas.

Performance evaluation

In Linearizability both reads & writes are blocking making things quite slow and difficult to manage in terms of failures. Again such a consistency model would not be applicable for a system that requires quick responses. You have to assume a good amount of latency with linearizability especially at scale. The difference was apparent even with 3 replicas.

Testing

Adding random delays to read & write operations is a way that I have tested my system. You can find various test cases in the ‘test_case.json’ file for Linearizability.

Please give the test cases ample time to run - one of the test cases takes nearly 40 seconds to complete a run for 5 replicas and 21 clients to simulate random delays that have been put in to imitate real world randomness in the network.

Eventual Consistency:

Implementation

I've stuck with the simplest design discussed in the lectures that does not involve any ordering of events.

For a write & read:

A client can connect to any replica and issue a SET or GET request. The replica sends back an acknowledgement after making an entry in the store for a write & for a read the replica acknowledges with the result of the current value of that key.

Performance evaluation

The reads and writes are fast. But the values might not be the latest values - a trade-off that can be made for speed & availability.

Testing

You can find various test cases in the 'test_case.json' file for Eventual consistency.