Multi Consistency KV – Report

This assignment implements multiple consistency models on a distributed key value store. Program can be started with some consistency.

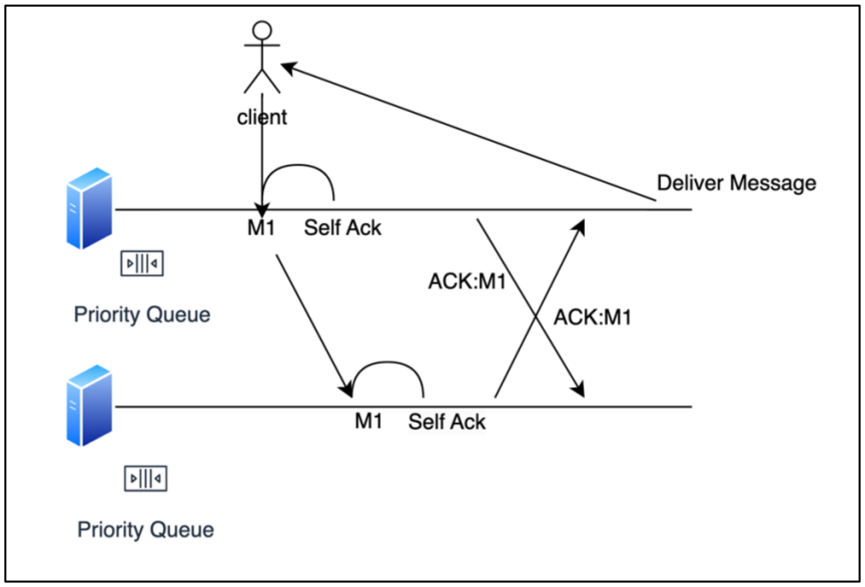
Key words and phrases: Total order broadcast: all acks, Logical clocks, Linearizability, Sequential consistency, Causal Consistency, Eventual Consistency.

1 Introduction

This implementation majorly uses total order broadcast for consensus. Implementation is started with highest consistency model – Linearizability. Rest of the consistency models are relaxation of few rules in the strong consistency model. *Total order broadcast: all acknowledgements* algorithm is used.

2 Design

2.1 Overview



Total Order Broadcast: All Acknowledgements

Only two servers are specified in the figure to reduce the complexity of the messages. Implementation can create up to five servers and tested with at least 3 servers and 2 clients.

2.2 Algorithm

- All messages are timestamped according to the consistency model.
- Message is broadcasted to all the nodes including itself.
- On message receive at a node:
    - Puts the message in the priority queue and broadcasts ack (including itself).
    - Queue is ordered based on time stamps.
- Message is delivered when it is at the head of the queue and all acks are received.

2.2 Correctness

This algorithm correctness depends on the *FIFO* ordering of messages. Basic assumption is that whenever there are concurrent messages, any node before sending the acknowledgement for the message from other server, it must send its own message.

- FIFO ordering is ensured by the underlying *TCP mechanism* which is used to send and receive messages.
- *Priority queue* ensures that everyone in the system is delivering the messages in same order after receiving all acknowledgements.
- Total order is ensured by using the server *port ids* to break the ties.

3 Implementation

Overview: Implementation uses Golang, Redis kv store and supports concurrency & parallelism. We can spawn up to 5 servers. Each server has server-server interface and client-server interface.

3.1 Linearizability

Linearizability ensures that every message is delivered to the application using some global wall clock timestamp ordering.

1. System wall clock time, with millisecond accuracy, is used to timestamp messages.
2. Even if there are ties with wall clock times, server port ids are used for total order.
3. Both reads and writes require consensus on the order of delivery. Slow reads/writes.

Reads	Writes
Blocking (Total order broadcast)	Blocking (Total order broadcast)

Message Fields & Client API

Unique Id*	:	Generated by server for each request.
Timestamp*	:	Unix epoch from the system is used to order. Ex: 1682991756242
Total order*	:	For total ordering we append server port. Ex: 1682991756242.59090
Op	:	Operation type (set/get)
Key	:	Key to be written or fetched
Value	:	Value to be written

Fields indicated by * are set by the server.

Example:

```
var message = &map[string]string {
   "id*": "1234959959", // unique id for each request
   "timestamp*": "38884940", // according to implementation
   "totalOrderTimestamp*": "38884940.1234"
   "op": "set", // set or get
   "key": "x",
   "value": "1",
}
```

Client sends server a TCP message in the above format. Set and get requests are differentiated by the "op" field. IDs are set by the server.

Server-Server vs Client-Server Communication

Each server listens on two separate ports and registers two threads. One for handling the connections from a client and another to handle internal communications.

Priority Queue

Priority queue orders messages according to the timestamps. Lower timestamp gets delivered first. Implementation is in "utils/pq.go". This guarantees that everyone processes the same order.

KV Store

Redis key-value store is used as internal data storage. This is used instead of the simple-kv designed in assignment 1 because simple-kv implementation uses persistent storage and limits testing for performance. Additionally, simple-kv implementation uses indexed-file system design and gets skewed and complex over time.
Redis key-value store is available on the Luddy servers. This implementation doesn't work without Redis client. More information about execution is given in "Build" section.

Message Delays

Simulating broadcast delays is done as follows:

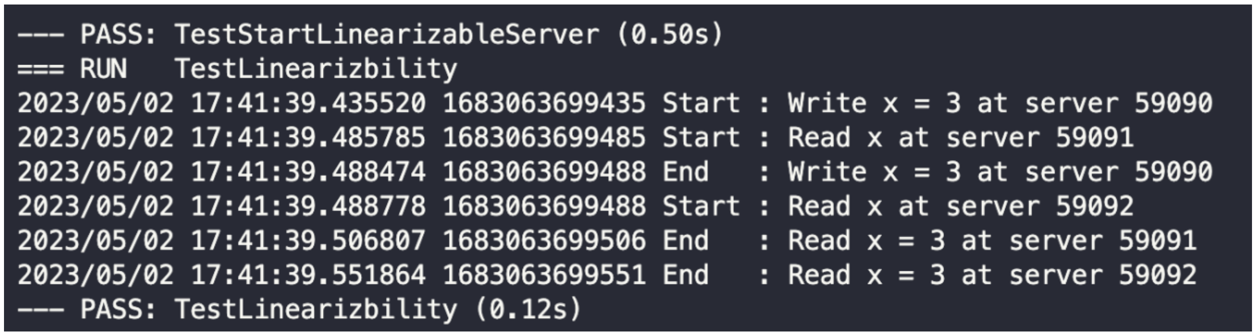
- Broadcast message function takes the amount of delay as an argument.
- Whenever message needs to be broadcasted. Additionally, it calculates distance from sending node to the receiving node.
- Before sending, each server sleeps for a small exponential time as delay^distance.
- Example: delay = 5ms.
    - Delay from server 1 to server 2 = 5^1 = 5ms
    - Delay from server 1 to server 3 = 5^2 = 25ms

This kind of delay mechanism facilitates testing concurrent writes in different systems and order of applying them. This also simulates real world latency, that depends on server-to-server distance.

Linearizability Tests:

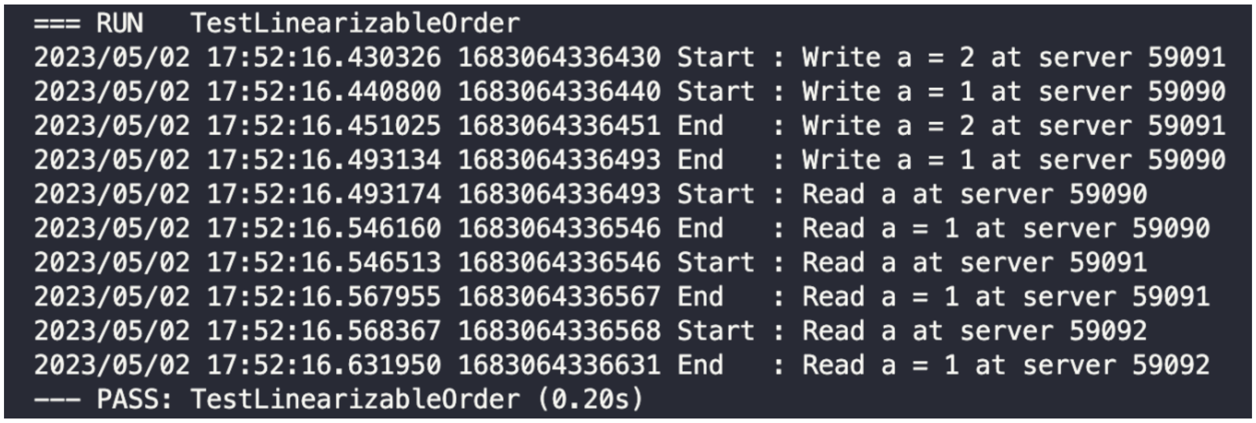
Running tests: $*make test-linearizability*

Test 1:
Blocking read writes.
Ordering according to the linearizable timestamps (Ex: 1683063699506 Milliseconds epoch)



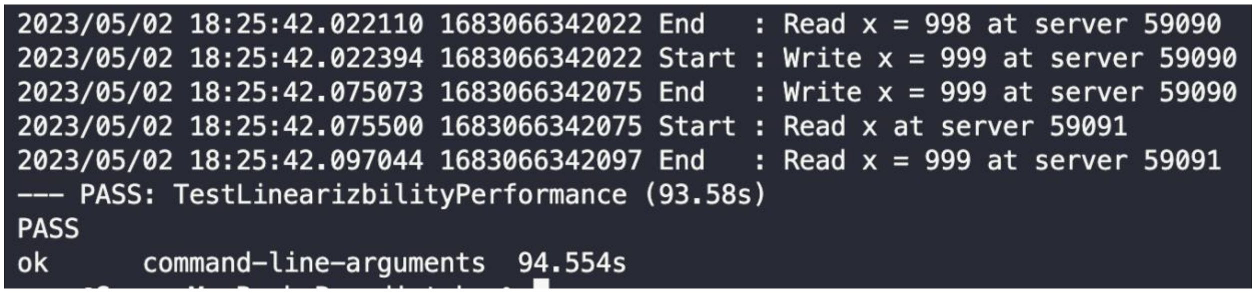
```
--- PASS: TestStartLinearizableServer (0.50s)
=== RUN   TestLinearizbility
2023/05/02 17:41:39.435520 1683063699435 Start : Write x = 3 at server 59090
2023/05/02 17:41:39.485785 1683063699485 Start : Read x at server 59091
2023/05/02 17:41:39.488474 1683063699488 End   : Write x = 3 at server 59090
2023/05/02 17:41:39.488778 1683063699488 Start : Read x at server 59092
2023/05/02 17:41:39.506807 1683063699506 End   : Read x = 3 at server 59091
2023/05/02 17:41:39.551864 1683063699551 End   : Read x = 3 at server 59092
--- PASS: TestLinearizbility (0.12s)
```

Test 2:
Order of applying operations is same at every server. (Both reads and writes)



```
=== RUN   TestLinearizableOrder
2023/05/02 17:52:16.430326 1683064336430 Start : Write a = 2 at server 59091
2023/05/02 17:52:16.440800 1683064336440 Start : Write a = 1 at server 59090
2023/05/02 17:52:16.451025 1683064336451 End   : Write a = 2 at server 59091
2023/05/02 17:52:16.493134 1683064336493 End   : Write a = 1 at server 59090
2023/05/02 17:52:16.493174 1683064336493 Start : Read a at server 59090
2023/05/02 17:52:16.546160 1683064336546 End   : Read a = 1 at server 59090
2023/05/02 17:52:16.546513 1683064336546 Start : Read a at server 59091
2023/05/02 17:52:16.567955 1683064336567 End   : Read a = 1 at server 59091
2023/05/02 17:52:16.568367 1683064336568 Start : Read a at server 59092
2023/05/02 17:52:16.631950 1683064336631 End   : Read a = 1 at server 59092
--- PASS: TestLinearizableOrder (0.20s)
```

Test 3: Performance test
Making 1000 Sequential requests: Reading own writes.



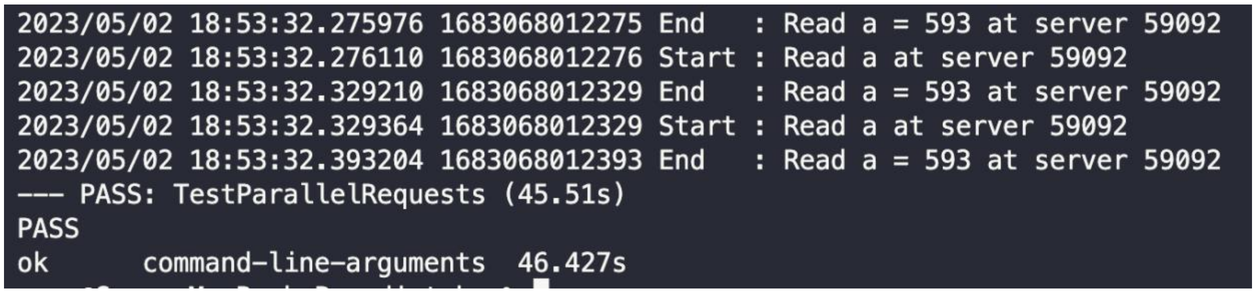
```
2023/05/02 18:25:42.022110 1683066342022 End   : Read x = 998 at server 59090
2023/05/02 18:25:42.022394 1683066342022 Start : Write x = 999 at server 59090
2023/05/02 18:25:42.075073 1683066342075 End   : Write x = 999 at server 59090
2023/05/02 18:25:42.075500 1683066342075 Start : Read x at server 59091
2023/05/02 18:25:42.097044 1683066342097 End   : Read x = 999 at server 59091
--- PASS: TestLinearizbilityPerformance (93.58s)
PASS
ok      command-line-arguments  94.554s
```

Performance

Performance is low relative to the loose consistency models because every operation requires a total order broadcast. As we can see from test 3 it took 95 seconds to process all the sequential read-my-own-write requests.

Even if the requests are made in parallel, to satisfy the most recent read (from tests 1 & 2) according to the global clock, each server processes them according to the priority queue.

Following is the performance when we make 1000 parallel connections. Each with write & read. Here we cannot expect to read what the program has written but can expect the latest value.



```
2023/05/02 18:53:32.275976 1683068012275 End   : Read a = 593 at server 59092
2023/05/02 18:53:32.276110 1683068012276 Start : Read a at server 59092
2023/05/02 18:53:32.329210 1683068012329 End   : Read a = 593 at server 59092
2023/05/02 18:53:32.329364 1683068012329 Start : Read a at server 59092
2023/05/02 18:53:32.393204 1683068012393 End   : Read a = 593 at server 59092
--- PASS: TestParallelRequests (45.51s)
PASS
ok      command-line-arguments  46.427s
```

2000/46 ~ 43 connections per sec
Note: These tests can fail on Luddy servers due to bandwidth usage and wait times.

Challenges & Improvements

- Total order broadcast: Improved by > 50% consensus (like Paxos).
- Consistency can be relaxed: Next sections.
- Broadcasts may fail: Fault tolerance.
- Clock drift at each server: Clock synchronization.
- Practicality: This implementation of consistency although highly desirable, but not practical.

3.2 Sequential Consistency

Implementation of sequential consistency is like the above version (same algorithm). But with *two relaxations*.

- Logical timestamps are used instead of wall clock time.
    - Each server's clock is updated as max{ sender's, receiver's clock}
- Reads no longer require consensus. Hence, they are local to the process.

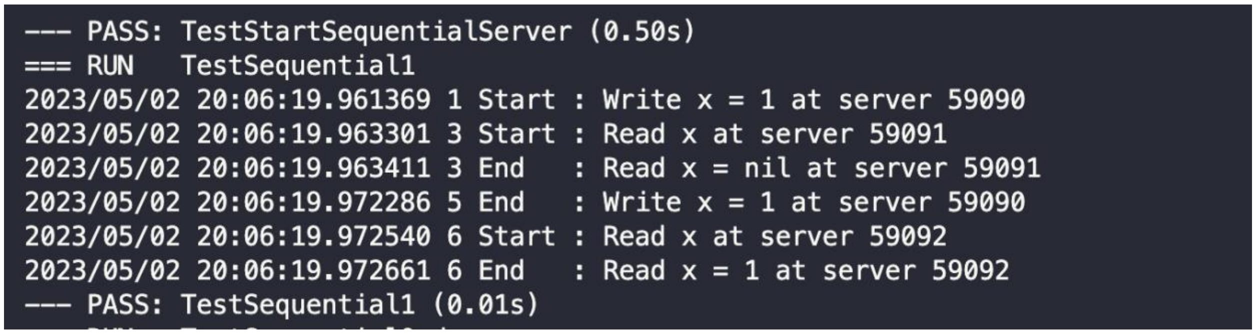
Reads	Writes
Local (Non-blocking)	Blocking (Total order broadcast)

Messages are like that of linearized version but now uses a Lamport timestamp.

Sequential Tests:

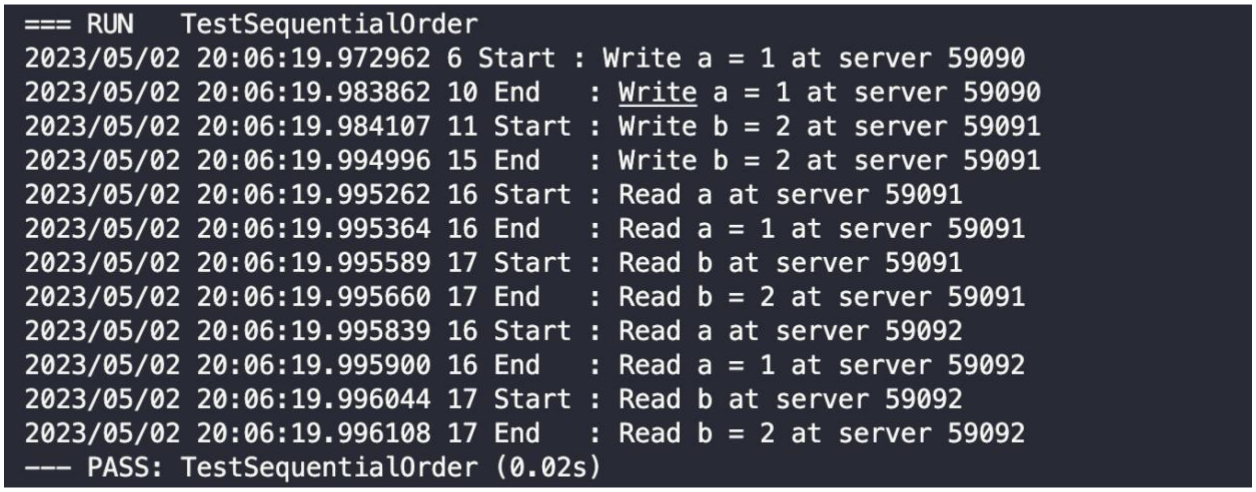
Running tests: $*make test-sequential*

Test 1:
Reads are non-blocking and do not require broadcast.
Lamport clock numbers are after the timestamp at each line.



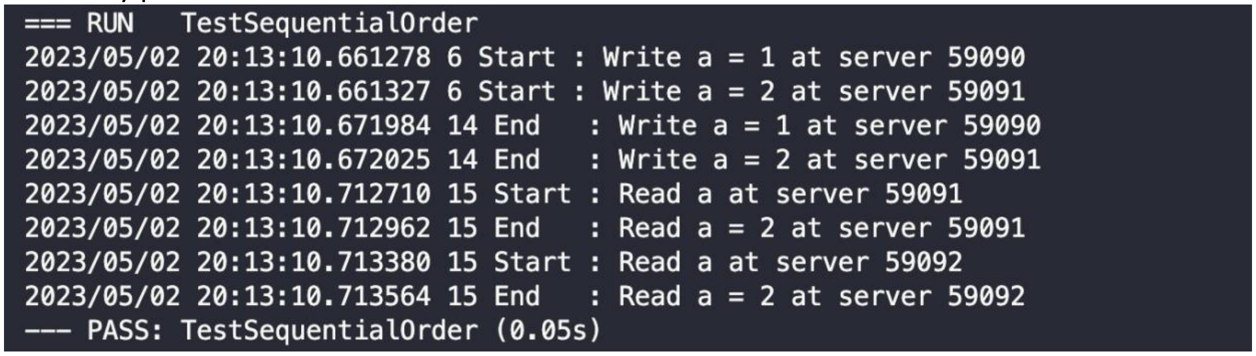
```
--- PASS: TestStartSequentialServer (0.50s)
=== RUN   TestSequential1
2023/05/02 20:06:19.961369 1 Start : Write x = 1 at server 59090
2023/05/02 20:06:19.963301 3 Start : Read x at server 59091
2023/05/02 20:06:19.963411 3 End   : Read x = nil at server 59091
2023/05/02 20:06:19.972286 5 End   : Write x = 1 at server 59090
2023/05/02 20:06:19.972540 6 Start : Read x at server 59092
2023/05/02 20:06:19.972661 6 End   : Read x = 1 at server 59092
--- PASS: TestSequential1 (0.01s)
```

Test 2:
Order of write operations must be same at every node. Follows logical clock.



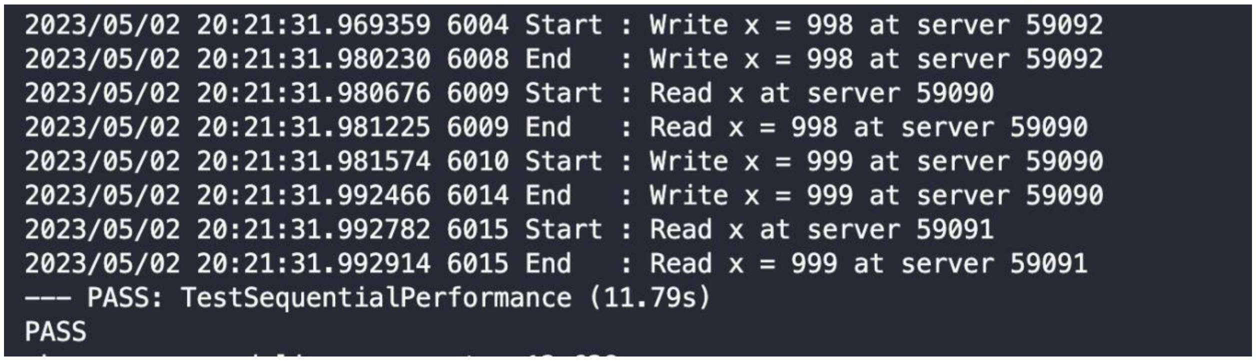
```
=== RUN   TestSequentialOrder
2023/05/02 20:06:19.972962 6 Start : Write a = 1 at server 59090
2023/05/02 20:06:19.983862 10 End   : Write a = 1 at server 59090
2023/05/02 20:06:19.984107 11 Start : Write b = 2 at server 59091
2023/05/02 20:06:19.994996 15 End   : Write b = 2 at server 59091
2023/05/02 20:06:19.995262 16 Start : Read a at server 59091
2023/05/02 20:06:19.995364 16 End   : Read a = 1 at server 59091
2023/05/02 20:06:19.995589 17 Start : Read b at server 59091
2023/05/02 20:06:19.995660 17 End   : Read b = 2 at server 59091
2023/05/02 20:06:19.995839 16 Start : Read a at server 59092
2023/05/02 20:06:19.995900 16 End   : Read a = 1 at server 59092
2023/05/02 20:06:19.996044 17 Start : Read b at server 59092
2023/05/02 20:06:19.996108 17 End   : Read b = 2 at server 59092
--- PASS: TestSequentialOrder (0.02s)
```

Here we can see there are ties with two write operations (started at same logical time) and it is broken by port number.



```
=== RUN   TestSequentialOrder
2023/05/02 20:13:10.661278 6 Start : Write a = 1 at server 59090
2023/05/02 20:13:10.661327 6 Start : Write a = 2 at server 59091
2023/05/02 20:13:10.671984 14 End   : Write a = 1 at server 59090
2023/05/02 20:13:10.672025 14 End   : Write a = 2 at server 59091
2023/05/02 20:13:10.712710 15 Start : Read a at server 59091
2023/05/02 20:13:10.712962 15 End   : Read a = 2 at server 59091
2023/05/02 20:13:10.713380 15 Start : Read a at server 59092
2023/05/02 20:13:10.713564 15 End   : Read a = 2 at server 59092
--- PASS: TestSequentialOrder (0.05s)
```

Test 3:
Making 1000 sequential requests to check program order of *read-your-own-writes*. This should break if the program does not read its own write in sequence.

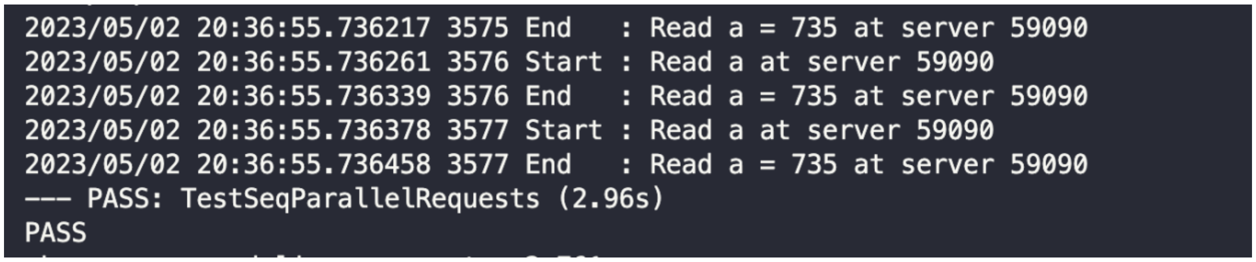


```
2023/05/02 20:21:31.969359 6004 Start : Write x = 998 at server 59092
2023/05/02 20:21:31.980230 6008 End   : Write x = 998 at server 59092
2023/05/02 20:21:31.980676 6009 Start : Read x at server 59090
2023/05/02 20:21:31.981225 6009 End   : Read x = 998 at server 59090
2023/05/02 20:21:31.981574 6010 Start : Write x = 999 at server 59090
2023/05/02 20:21:31.992466 6014 End   : Write x = 999 at server 59090
2023/05/02 20:21:31.992782 6015 Start : Read x at server 59091
2023/05/02 20:21:31.992914 6015 End   : Read x = 999 at server 59091
--- PASS: TestSequentialPerformance (11.79s)
PASS
```

Performance:

Performance is relatively high compared to the linearizable consistency, as the read operations are non-blocking. As we can observe, test case 3 took around 12 seconds which is more than *85% improvement (less time)* compared to the linearizable sequential test (it took 94 seconds). We should at least expect 50% improvement because now almost every read is instantaneous.

Following is the performance when we make 1000 parallel connections. Each with write & read. Here we cannot expect to read what the program has written but can expect the latest value.



```
2023/05/02 20:36:55.736217 3575 End   : Read a = 735 at server 59090
2023/05/02 20:36:55.736261 3576 Start : Read a at server 59090
2023/05/02 20:36:55.736339 3576 End   : Read a = 735 at server 59090
2023/05/02 20:36:55.736378 3577 Start : Read a at server 59090
2023/05/02 20:36:55.736458 3577 End   : Read a = 735 at server 59090
--- PASS: TestSeqParallelRequests (2.96s)
PASS
```

2000 parallel requests (read + write) just took around 3 seconds to complete. Which implies 666 req/sec.
Note: These tests can fail on Luddy servers due to bandwidth usage and wait times.

Challenges & Improvements

- Consensus: Similarly, even here we can improve with > 50% consensus
- Scaling: Total order broadcast and blocking-writes cannot scale. We can improve using causal consistency, which is provided next.
- Broadcasts may fail: Fault Tolerance.
- Practicality: Not used unless strictly necessary. Most practical systems are strong eventual.

3.3 Causal Consistency

Implementation of causal consistency relaxes consensus on ordering of events. Hence, servers don’t need to “fix on some order “ of events. But causal consistency requires ordering “potentially causal events”.

Non-Primary based approach (Broadcast with dependencies)

1. As writes can go to any node, we need to make sure that causally related writes are applied correctly at each replica.
2. This is done by injecting dependency in the message from the client every time a causal write event occurs.
3. Reads are like the primary based approach, where clients send minimum acceptable version.

Reads	Writes
Kind of Fast (sometimes blocking)	Non-Blocking

Message Formats

Client Write:

```
var message = &map[string]string {
   "op": "set",
   "key": "x",
   "value": "2",
   "dependency": "{ "key": "x", "version": "1"}",
}
```

These dependencies ensure that whenever there is a broadcast for some write operation, before applying the new write, replica waits until the dependent write is done.

Client Read:

```
var message = &map[string]string {
   "op": "set",
   "key": "x",
   "minVersion": "2",
}
```

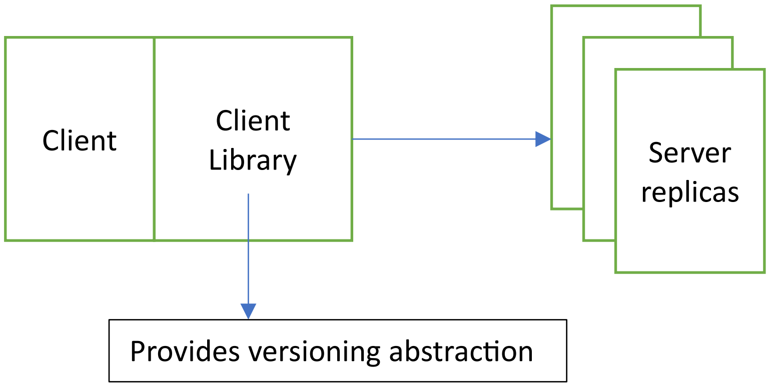
Minimum version field specifies that the replica must wait until it writes the minimum acceptable version.

Relaxations

- Unless there are dependencies, replicas can apply them in any order.
- Write are non-blocking and fast.
- Without minimum version restriction, reads are local and very fast.

Client Interface Abstraction:

In this implementation client interface provides an *abstraction for all the versioning* and causal order consistency. Hence, client programs do not explicitly require mentioning the version they want. We can enable versioning in the client library to capture causal events!
Implementation is in "services/client.go" and "kv_causal_test.go"



Correctness

Suppose a client sends a Wr(x=1) dependency Rd(x, version 0).
Now the same client sends a Wr(x = k >= 1) with dependency Rd(x, version k - 1). All replicas only write x = k only when version k – 1 is applied.
Now for Wr(x = k+1) implies Wr(x = k) is already applied.

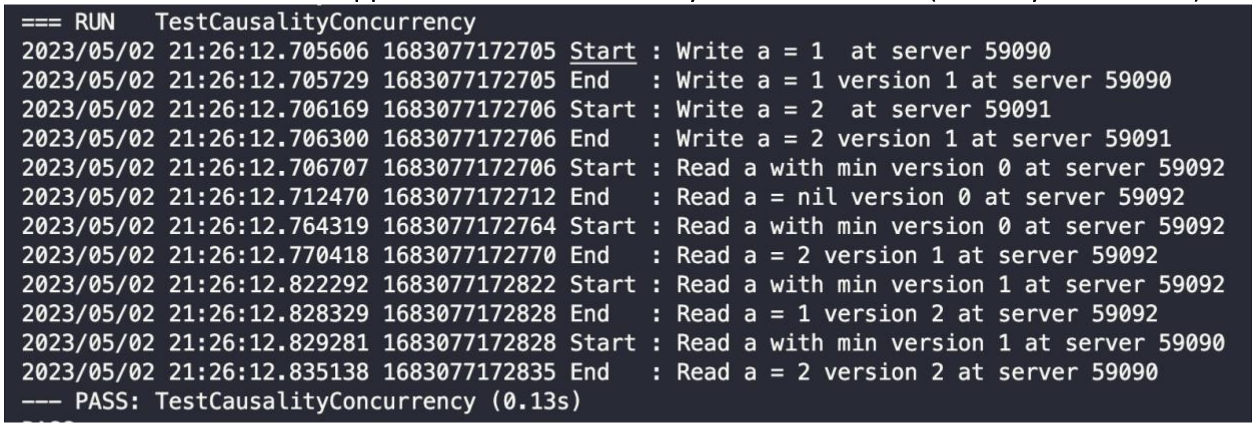
Versioning

Every time a replica sees a new request for the existing key, it increments its version and stores the new value. Version numbers are independent at each replica. This is not required for causality. Ex: Concurrent writes on same key can have same version numbers at different replicas. Since concurrent writes are not causal, this is ok.

Causality Tests:

How to run: *$make test-causal*

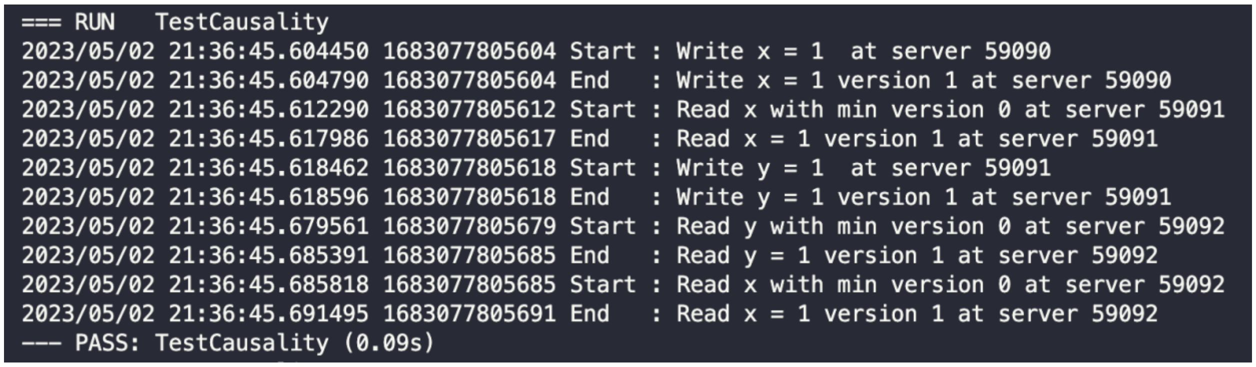
Test 1: (Read causality + Concurrent writes)
Concurrent writes can be applied in different order by different clients. (Causally nonrelated)



```
=== RUN   TestCausalityConcurrency
2023/05/02 21:26:12.705606 1683077172705 Start : Write a = 1  at server 59090
2023/05/02 21:26:12.705729 1683077172705 End   : Write a = 1 version 1 at server 59090
2023/05/02 21:26:12.706169 1683077172706 Start : Write a = 2  at server 59091
2023/05/02 21:26:12.706300 1683077172706 End   : Write a = 2 version 1 at server 59091
2023/05/02 21:26:12.706707 1683077172706 Start : Read a with min version 0 at server 59092
2023/05/02 21:26:12.712470 1683077172712 End   : Read a = nil version 0 at server 59092
2023/05/02 21:26:12.764319 1683077172764 Start : Read a with min version 0 at server 59092
2023/05/02 21:26:12.770418 1683077172770 End   : Read a = 2 version 1 at server 59092
2023/05/02 21:26:12.822292 1683077172822 Start : Read a with min version 1 at server 59092
2023/05/02 21:26:12.828329 1683077172828 End   : Read a = 1 version 2 at server 59092
2023/05/02 21:26:12.829281 1683077172828 Start : Read a with min version 1 at server 59090
2023/05/02 21:26:12.835138 1683077172835 End   : Read a = 2 version 2 at server 59090
--- PASS: TestCausalityConcurrency (0.13s)
```

As we can see that value of a, where version 2 of the same is read as a = 1 at server 59092 and read as a = 2 at server 59090. This is not true in sequential consistency!
This test case can also verify minimum read version that a client specifies for read causality.

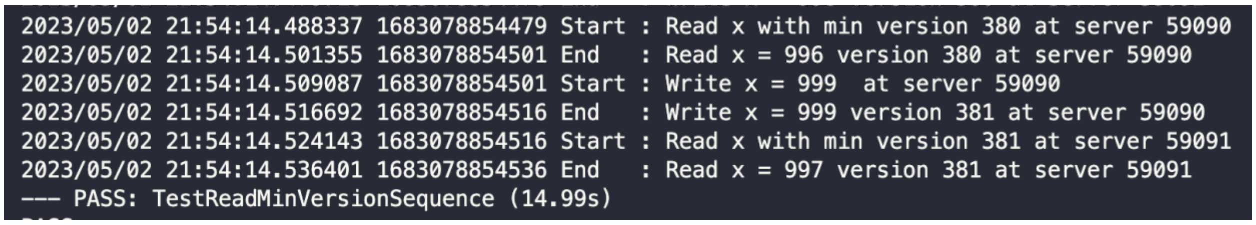
Test 2: (Write causality)
Testing causally related events. Let the servers be S1, S2 and S3.
Test case: S1:Wr(x=1), S2:Rd(x=1), S2:Wr(y=1), S3:Rd(y=1), S3:Rd(x=1). Here if server S3 reads y as 1 it must read x = 1 next. This test case can be *run many times* to check if the kv-store is causally consistent.



```
=== RUN   TestCausality
2023/05/02 21:36:45.604450 1683077805604 Start : Write x = 1  at server 59090
2023/05/02 21:36:45.604790 1683077805604 End   : Write x = 1 version 1 at server 59090
2023/05/02 21:36:45.612290 1683077805612 Start : Read x with min version 0 at server 59091
2023/05/02 21:36:45.617986 1683077805617 End   : Read x = 1 version 1 at server 59091
2023/05/02 21:36:45.618462 1683077805618 Start : Write y = 1  at server 59091
2023/05/02 21:36:45.618596 1683077805618 End   : Write y = 1 version 1 at server 59091
2023/05/02 21:36:45.679561 1683077805679 Start : Read y with min version 0 at server 59092
2023/05/02 21:36:45.685391 1683077805685 End   : Read y = 1 version 1 at server 59092
2023/05/02 21:36:45.685818 1683077805685 Start : Read x with min version 0 at server 59092
2023/05/02 21:36:45.691495 1683077805691 End   : Read x = 1 version 1 at server 59092
--- PASS: TestCausality (0.09s)
```

At the end (last but one line) even though there is no minimum version requirement at server 59092, x is read as 1.

Test 3: (Read latest write by specifying versions)



```
2023/05/02 21:54:14.488337 1683078854479 Start : Read x with min version 380 at server 59090
2023/05/02 21:54:14.501355 1683078854501 End   : Read x = 996 version 380 at server 59090
2023/05/02 21:54:14.509087 1683078854501 Start : Write x = 999  at server 59090
2023/05/02 21:54:14.516692 1683078854516 End   : Write x = 999 version 381 at server 59090
2023/05/02 21:54:14.524143 1683078854516 Start : Read x with min version 381 at server 59091
2023/05/02 21:54:14.536401 1683078854536 End   : Read x = 997 version 381 at server 59091
--- PASS: TestReadMinVersionSequence (14.99s)
```

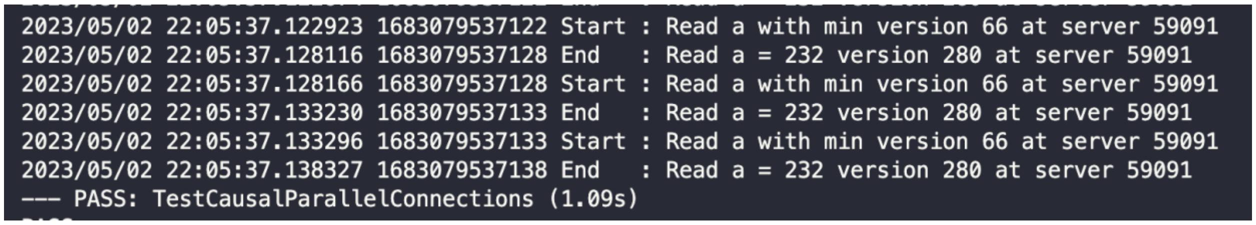
Here we can observe at the end that even though we wrote x = 999 with version 381 at server 59090. Since the concurrent write operations versioning is independent and server 59091's x is 997 when with version 381. It is trivial to note that write operations and read operations are performed on different servers.

Ironically, this test takes similar time to a sequentially consistent store because the reads here are not completely non-blocking. They wait when we specify minimum version number. We can further compare the performance with eventually consistent store.

Performance

Causal consistency without any causal dependencies is like eventual consistency and there are no wait times associated. This can be observed from the following performance test.

Test 4: (Performance)


```
2023/05/02 22:05:37.122923 1683079537122 Start : Read a with min version 66 at server 59091
2023/05/02 22:05:37.128116 1683079537128 End   : Read a = 232 version 280 at server 59091
2023/05/02 22:05:37.128166 1683079537128 Start : Read a with min version 66 at server 59091
2023/05/02 22:05:37.133230 1683079537133 End   : Read a = 232 version 280 at server 59091
2023/05/02 22:05:37.133296 1683079537133 Start : Read a with min version 66 at server 59091
2023/05/02 22:05:37.138327 1683079537138 End   : Read a = 232 version 280 at server 59091
--- PASS: TestCausalParallelConnections (1.09s)
```

This test is made for 500 concurrent connections, each with both read and write. This program took around 1 second for 1000 requests ~ 1000 req/sec.
We can observe here that using causal consistency we can exploit nonrelated ordering of events and gain good performance overall. The performance gain more than 30% compared to the sequential consistency implementation.

VS Primary based approach

Primary based approach requires writes to be forwarded to a single server. This makes primary a single point of failure and limits the performance gain we achieved from this implementation!

Challenges & Improvements:

- Dependency on Clients: This program requires for clients to explicitly mention the dependencies for a replica to know.
- Client Library: For the above-mentioned reason this method might require client level abstractions to provide causally consistent reads and writes.
- Perspective of Causality: Humans always perceive causality differently. For example, we can think of writing something and later letting know someone by a phone call. This *out of band causality* cannot be maintained in the system.
- Partial ordering: As we observed causal consistency do not limit ordering of events.
- Fast Reads: If everything is causally related reads take similar time to the sequential one. Eventual Consistency can help!

3.4 Eventual Consistency

When there are no concurrent writes, replicas become eventually consistent. Everything is relaxed now!

- No requirement on ordering of events, causality etc.
- Super-fast!
- Hope is a good thing for eventual consistency!

Writes	Reads
Non-Blocking (Blazing fast!)	Non-Blocking (Blazing fast!)

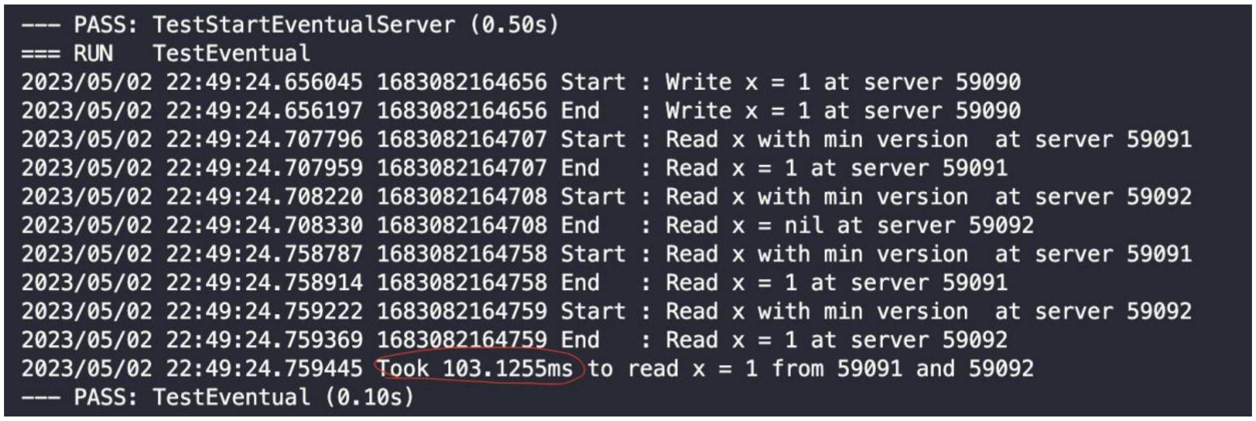
Implementation

Background broadcast for writes
Resolutions are asynchronously done at background
Local Reads
Local Writes

Tests

$make test-eventual

Test 1: (Eventually read the same value)
In this test we write x=1 to replica 1 and wait until we read the same value from replicas 2 and 3.



```
--- PASS: TestStartEventualServer (0.50s)
=== RUN   TestEventual
2023/05/02 22:49:24.656045 1683082164656 Start : Write x = 1 at server 59090
2023/05/02 22:49:24.656197 1683082164656 End   : Write x = 1 at server 59090
2023/05/02 22:49:24.707796 1683082164707 Start : Read x with min version  at server 59091
2023/05/02 22:49:24.707959 1683082164707 End   : Read x = 1 at server 59091
2023/05/02 22:49:24.708220 1683082164708 Start : Read x with min version  at server 59092
2023/05/02 22:49:24.708330 1683082164708 End   : Read x = nil at server 59092
2023/05/02 22:49:24.758787 1683082164758 Start : Read x with min version  at server 59091
2023/05/02 22:49:24.758914 1683082164758 End   : Read x = 1 at server 59091
2023/05/02 22:49:24.759222 1683082164759 Start : Read x with min version  at server 59092
2023/05/02 22:49:24.759369 1683082164759 End   : Read x = 1 at server 59092
2023/05/02 22:49:24.759445 Took 103.1255ms to read x = 1 from 59091 and 59092
--- PASS: TestEventual (0.10s)
```

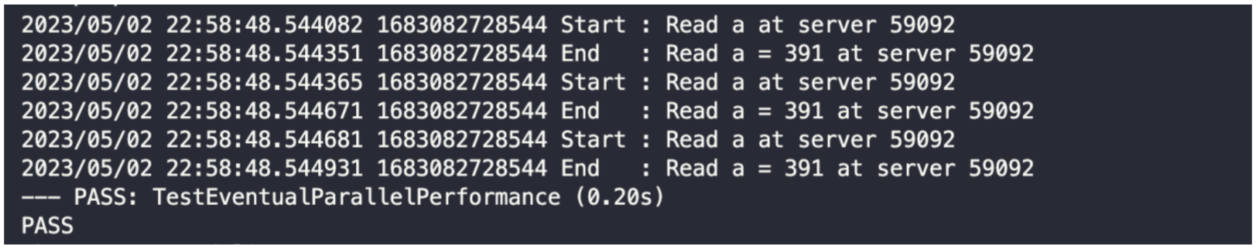
As we can see, with the injected exponential delays, 10^1 and 10^2 (10 and 100ms) it took around 103 milliseconds for replicas to converge when there are no concurrent writes!

Performance

Eventual consistency is highly performant because it doesn't care about any strong conditions. We can see how parallel requests are responded in the below test case.

Test 2 (Performance):

500 parallel connections with both read and write requests.



```
2023/05/02 22:58:48.544082 1683082728544 Start : Read a at server 59092
2023/05/02 22:58:48.544351 1683082728544 End   : Read a = 391 at server 59092
2023/05/02 22:58:48.544365 1683082728544 Start : Read a at server 59092
2023/05/02 22:58:48.544671 1683082728544 End   : Read a = 391 at server 59092
2023/05/02 22:58:48.544681 1683082728544 Start : Read a at server 59092
2023/05/02 22:58:48.544931 1683082728544 End   : Read a = 391 at server 59092
--- PASS: TestEventualParallelPerformance (0.20s)
PASS
```

As we can see the blazing fast performance by trading off consistency. It just took 20ms to process 1000 requests ~ 5000 req/sec. This is **5x** the causal consistency!

Challenges & Improvements:

- Convergence: During concurrent write convergence can be provided by strong eventual consistency using CRDTs
- Faults: Internal faults can lose data. Like broadcast failures extra. As these are asynchronous, users cannot know.
- Periodic checking: Cron jobs may need to periodically check the data for consistency issues
- Faults: Fault Tolerance.
- … more (Leads to strong eventual consistency)

4 Build

4.1 Makefile

$make test-linearizability
$make test-sequential
$make test-causal
$make test-eventual

4.2 Environment

Go version 1.13 is already on Luddy servers. It is possible that it may ask for newer version.
Go Redis client package is required to connect with Redis.
Redis server is already installed on Luddy servers.
Dockerfile is provided for such scenarios.

4.3 Directory Structure

- Services
  - All consistency models implementations
- Utils
  - Priority queue, config loading & redis client helper
- All tests are in main folder
- Makefile
- Config (JSON file can be edited for config settings like num servers, ports etc.)
- Dockerfile

To manually run tests please look at the Makefile.
If for some reason this could not be run Luddy servers. We need Redis and Golang to be installed on the system.
Redis: https://redis.io/download/
Golang: https://go.dev/dl/
Repo Link: https://github.iu.edu/vaganj/dist-kv/
Docker file is also written. We can run this program in Luddy docker supported servers using the command *$docker build -t app .*