Project 2 - KVS
Amirmohammad Nazari

—

Linearizability Test

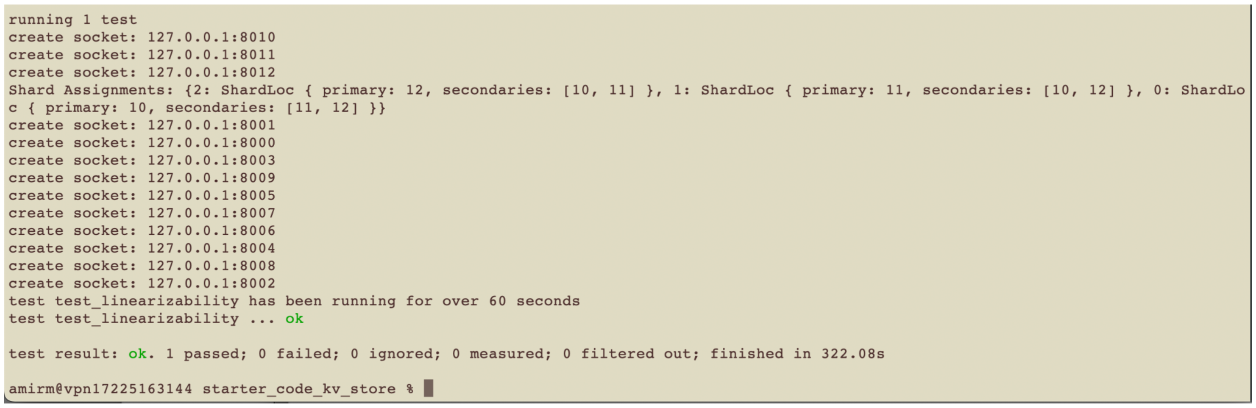
For this test, my approach involves establishing a globally ordered sequence of operations, following the assumptions outlined in the instructions. To achieve this, I undertake the following steps:

1. I filter all the PUT operations and perform an exhaustive search to determine the correct order in which they are executed. This is done by checking each new PUT observed value like K must correspond to a PUT with new value K. This approach ensures linearizability for PUTs since we can confidently state that each PUT operation introduces a new value for its corresponding key.

2. For each GET operation, I search within the ordered PUT operations to ensure that a valid sequence allows the GET operation to follow at least one preceding PUT operation.

Note that this implementation of linearizability check is very slow. It actually takes 5-6 minutes to run the *linearizability.rs* test, which can be seen in the following figure. This figure shows one run of the following command:

```
cargo test --test linearizability
```



```
running 1 test
create socket: 127.0.0.1:8010
create socket: 127.0.0.1:8011
create socket: 127.0.0.1:8012
Shard Assignments: {2: ShardLoc { primary: 12, secondaries: [10, 11] }, 1: ShardLoc { primary: 11, secondaries: [10, 12] }, 0: ShardLo
c { primary: 10, secondaries: [11, 12] }}
create socket: 127.0.0.1:8001
create socket: 127.0.0.1:8000
create socket: 127.0.0.1:8003
create socket: 127.0.0.1:8009
create socket: 127.0.0.1:8005
create socket: 127.0.0.1:8007
create socket: 127.0.0.1:8006
create socket: 127.0.0.1:8004
create socket: 127.0.0.1:8008
create socket: 127.0.0.1:8002
test test_linearizability has been running for over 60 seconds
test test_linearizability ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 322.08s

amirm@vpn17225163144 starter_code_kv_store %
```

Benchmark Results

Throughput (Kops):

	Low Contention	High Contention
Read-mostly	84.47	75.84
Write-mostly	28.12	23.81

Avg Latency (us):

	Low Contention	High Contention
Read-mostly	94.01	104.83
Write-mostly	284.36	335.24

Median (us):

	Low Contention	High Contention
Read-mostly	84	93
Write-mostly	285	320

95% (us):

	Low Contention	High Contention
Read-mostly	177	209
Write-mostly	385	548

99% (us):

	Low Contention	High Contention
Read-mostly	282	297
Write-mostly	438	746

- In general, read operations are typically faster and less resource-intensive compared to write operations. In our case, we are using a hashmap data structure, and for write operations, there might be a need to modify the hashmap's structure, which can be a computationally expensive process. More importantly, with get operations no communication and synchronization among the servers is necessary, whereas for each put all servers must communicate and replicate transactions.

  As a result, it is evident that read-mostly benchmarks tend to outperform write-mostly benchmarks. For example, when comparing the two scenarios, we observed a substantial difference in performance metrics. The throughput decreased by 66%, and the average latency and median response times increased by 200%. Similar trends can be observed in the 95th and 99th percentile tail latencies, which increased by 117% and 55%, respectively. The same argument holds for both low- and high-contention settings, and the 95 and 99 tail latencies increase by 160% and 150%, respectively.

- In settings with low contention, where processes are not vying heavily for access to the same keys, average latency typically remains lower. This is primarily because requests can be processed more swiftly and with minimal interference from other concurrent operations. Conversely, in high contention scenarios, transactions can be delayed, hindering quick operations.

  This effect becomes particularly pronounced in write-heavy scenarios, where contention often emerges due to 'put' operations. Consequently, we observe that, as contention increases, average and median latencies in read-mostly scenarios increase by approximately 10%, while the increase in write-mostly scenarios is closer to 20%. This trend is more noticeable in the 95th and 99th percentile tail latencies, which increase by approximately 10-20%. However, in write-mostly scenarios, these latencies exhibit a more substantial increase, ranging from 50-70%.

  The impact on throughput is not significantly different between low- and high-contention scenarios. We can observe a reduction in throughput of approximately 10-20%.