Project 3 - KVS
Amirmohammad Nazari

—

Linearizability Test

***

I use the exact same approach as I used in the previous project to test linearizability. I will just use my explanations from the previous report with the new screenshot of this lab's 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
create socket: 127.0.0.1:8000
create socket: 127.0.0.1:8001
create socket: 127.0.0.1:8002
create socket: 127.0.0.1:8009
create socket: 127.0.0.1:8004
create socket: 127.0.0.1:8007
create socket: 127.0.0.1:8005
create socket: 127.0.0.1:8006
create socket: 127.0.0.1:8003
create socket: 127.0.0.1:8008
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 335.81s
```

Benchmark Results: No Cache vs. Directory-Based Caching

Throughput (Kops):

	No Cache		Directory-Based Cache Coherency	
	Low Contention	High Contention	Low Contention	High Contention
Read-mostly	31.0	30.4	48.9	61.2
Write-mostly	31.0	30.7	27.0	28.1

Avg Latency (us):

	No Cache		Directory-Based Cache Coherency	
	Low Contention	High Contention	Low Contention	High Contention
Read-mostly	252.5	254.7	161.8	129.7
Write-mostly	252.0	254.6	292.4	276.3

Median (us):

	No Cache		Directory-Based Cache Coherency	
	Low Contention	High Contention	Low Contention	High Contention
Read-mostly	254	249	130	87
Write-mostly	252	254	292	284

95% (us):

	No Cache		Directory-Based Cache Coherency	
	Low Contention	High Contention	Low Contention	High Contention
Read-mostly	335	334	396	399
Write-mostly	337	334	593	563

99% (us):

	No Cache		Directory-Based Cache Coherency	
	Low Contention	High Contention	Low Contention	High Contention
Read-mostly	368	425	540	551
Write-mostly	379	389	698	663

Important Note

Sometimes the servers fail because of this error:

```
thread '<unnamed>' panicked at 'attempt to calculate the remainder with a
divisor of zero', kv_store::kvs::get_shard_id_from_key, src/kvs.rs:129:5
```

The reason is that controller threads start at the same time with the clients and therefore, in some cases, the ***PutShardInfo*** messages won't reach the servers on time and client requests reach them sooner. Therefore, the server does not know how to find the home node of a key and throws an error message. I believe that the best solution to this problem is to put a Sleep() function after the controller threads start and before running the clients, and therefore, we need to modify the test codes.

Questions

- Based on the results shown above in the **read-mostly** situation, the throughput, average latency, and median latency have been improved compared to the no-caching case. The reason is that most operations here are GET operations and most of them will be responded to directly from the cache and won't be forwarded to the home node of the key. However, some operations here are more expensive than no-caching scenarios because they require multiple communication messages between servers to achieve cache coherency. Hence, the 95% and 99% tail latencies are much higher than the no-cache experiments.

  On the other hand, in the **write-mostly** scenarios, most of the operations are PUT and have to be forwarded to the home server of keys. As a result, we get similar results comparing no-cache but they are a little worse due to the extra communications between servers.

Finally, in contrast to the previous project, having more **contention** helps the DSM-based system because more and more queries will be responded to from the cache, and therefore, in all cases, the throughput increases and latencies drop.

- The throughput here in the read-only scenario is worse than in the previous lab because of the communication overhead of shared memory abstraction. In project 2 when we had read-mostly applications most of the queries would be answered directly but here we have to wait for coherence protocol to make sure the distributed memory is consistent.

- The API to shared memory abstraction is much easier to work with because the user will only see all the memory parts as one unified global memory and won't have to deal with the complexities of distributed memory. On the other hand, the user will have less access to optimize the system for themselves. In the message-passing approach, users can modify the underlying consistency protocol based on their needs and maybe choose to lose some sort of consistency to get better performance in their specific workload.

- Implementing the DSM approach is much harder than message passing as the system designer has to deal with all the inherent complexities of consistency of shared memory in a distributed setting. However, in message-passing systems system designers will not handle the consistency and racing issues and leave those up to the users.