Team info

Haorong Lu (hl80)
Jiaai Xu (jiaaixu2)
Lan Zhang (lz31)

Design and Implementation

Using the Multiple-Write-Multiple-Read (MWMR) shared register protocol as a basis, we developed two main programs in Golang: the client and the replica. The client program includes four important functions, namely GetPhase and SetPhase for the Writer and Reader, respectively, which send the corresponding RPC requests to the replica. We use gRPC as the communication protocol between clients and replicas. Since the GetPhase/SetPhase functions for both the Writer and Reader are almost identical in terms of RPC message format, we define only two gRPC methods: GetPhase and SetPhase. The replica's main responsibility is to act as a backend for these two gRPC methods and update its own local storage, which is represented by a global hash table (map in Go).

A challenging aspect of the clients was determining how to concurrently send gRPC requests to all replicas and wait for a majority of responses before proceeding. To accomplish this, we used a combination of goroutines, sync.WaitGroup, and channels. Below is a simplified version of our approach:

```
var wg sync.WaitGroup
wg.Add(n)
ch := make(chan Pair, n)

for i := 0; i < n; i++ {
	go func(rid int) {
		// Send a gRPC request and put the response in `temp`
		// ......
		ch <- temp
		wg.Done()
	}(i)
}

go func() {
	wg.Wait()
}()

done := make([]Pair, 0, f+1)
for pair := range ch {
	done = append(done, pair)
	if len(done) >= f+1 {
		break
	}
}
```

Another important consideration is that we use a global map as the replica storage, which is not thread-safe in Go. Since multiple gRPC streams may access the same key simultaneously, we need to add a RWLock to the map. To minimize the locking overhead, we added a fine-grained lock to each key-value pair instead of using a global lock. This design improves the replica's concurrency performance, especially when handling a large number of clients. The remaining parts were relatively straightforward, involving the translation of MWMR provisions into corresponding code. If you have any further questions about the design and implementation, you can refer directly to our source code.

Testing Correctness

Before any evaluation on the performance of our implementation, correctness needs to be promised first, which is tested as follows.

1. Start 5 servers on individual Bare Metal PCs with a data store of size 1000 (1000 key-value pairs). One replica is set to fail in the middle of the running.
   Bash Command: `./scripts/run_server.sh -s 1000`
2. Start 32 multiple clients concurrently on another cloud Bare Metal PC where each client is going to send 10000 reads and 10000 writes rpc requests to the distributed storage system. Then the test script will track every read and write event with its key, timestamp and value, then log it into a log file.
   Bash Command: `./scripts/client_test.sh -i 1000 -n 32 -r 10000 -w 10000 -c true`
   (-c means correctness mode, the program will skip useless event tracking operations. Here we choose 1000 as the data store size and 32 as the number of clients because they maximize the concurrency and thus maximizing the possibility of exposing the linearizability problem.)
3. Run a Python script called verify_correctness.py to verify the linearizability of each client's log file. If the script prints "Linearizability Verified!", then linearizability of all clients are verified and correctness is verified.
   Bash Command: `python3 data_processing/verify_correctness.py`

Logistics of the Verification Script

Under testing correctness mode, each client opens its own log file with its id as the unique identifier under the folder "logs_for_correctness". The log files format is as follows:

```
INFO: 2023/02/26 17:46:03 main.go:159: ithwrite: 300 key: 474 timestamp: 6 cid: 1 value: 791
INFO: 2023/02/26 17:46:03 main.go:146: ithread: 301 key: 529 timestamp: 7 cid: 14 value: 166
INFO: 2023/02/26 17:46:03 main.go:146: ithread: 302 key: 327 timestamp: 4 cid: 19 value: 320
INFO: 2023/02/26 17:46:03 main.go:146: ithread: 303 key: 857 timestamp: 8 cid: 29 value: 176
INFO: 2023/02/26 17:46:03 main.go:146: ithread: 304 key: 929 timestamp: 9 cid: 9 value: 994
INFO: 2023/02/26 17:46:03 main.go:159: ithwrite: 304 key: 876 timestamp: 6 cid: 1 value: 106
```

where 'key', 'timestamp', 'cid' and the relative order of each log data are needed for verification. As for one single client, it will send a new rpc request only after the previous one is finished so it

is promised that in one log file, the real-time order is the same as the order in the log file. Therefore, the only next step to verify the linearizability is to check that the timestamp is not decreasing for the same key. The Python script is shown below.

```
import os

def isT1BeforeT2(t1: list, t2: list):
   if t1[0] < t2[0]: return True
   elif t1[0] > t2[0]: return False
   else:
       if t1[1] <= t2[1]: return True
       else: return False

directory = "logs_for_correctness/logs_with_32_clients"
for filename in os.listdir(directory):
   data = {}
   filepath = os.path.join(directory, filename)
   if not os.path.isfile(filepath):
       print("not a file {}".format(filepath))
       continue
   f = open(filepath, 'r')
   for line in f.readlines():
       line = line.split()
       if line[7] not in data:
           data[line[7]] = [int(line[9]), int(line[11])]
       else:
           if isT1BeforeT2(data[line[7]], [int(line[9]), int(line[11])]): data[line[7]] =
[int(line[9]), int(line[11])]
           else:
               print("NOT linear!")
               print(filename,  "line: ", line)
               f.close()
               exit(1)
   f.close()

print("Linearizability Verified!")
```

Evaluation

General Setup

Hardware: 6 Bare Metal PCs, detailed information can be found in the table below

Type	m510
Class	pc
Architecture	x86_64
dom0mem	4096M
hw_cpu_bits	64
hw_cpu_cores	8
hw_cpu_hv	1
hw_cpu_sockets	1
hw_cpu_speed	2000
hw_cpu_threads	2
hw_mem_size	65536
processor	Intel Xeon D-1548
adminmfs_osid	ADMIN-FREEBSD-MFS
default_imageid	UBUNTU18-64-X86
default_osid	UBUNTU18-64-X86
delay_osid	FBSD102-64-STD
diskloadmfs_osid	FRISBEE-FREEBSD-MFS
jail_osid	FBSD102-64-STD
recoverymfs_osid	RECOVERY-LINUX

Latencies and Bandwidth

As we started 5 PCs as servers and 1 PC as concurrent clients, latencies and bandwidth between the 1 PC and the 5 PCs are essential. As measured and shown below, the average latency between a replica and the client is about 0.15 ms and the bandwidth is 9.28Gbits/sec, so

theoretically one client connected with five replicas should have bandwidth around 1.8Gbits/sec and 1.9Gbits/sec.

```
64 bytes from 128.110.217.121: icmp_seq=1 ttl=64 time=0.148 ms
64 bytes from 128.110.217.121: icmp_seq=2 ttl=64 time=0.158 ms
64 bytes from 128.110.217.121: icmp_seq=3 ttl=64 time=0.153 ms
64 bytes from 128.110.217.121: icmp_seq=4 ttl=64 time=0.149 ms
64 bytes from 128.110.217.121: icmp_seq=5 ttl=64 time=0.154 ms
64 bytes from 128.110.217.121: icmp_seq=6 ttl=64 time=0.155 ms
```

```
[  3] local 128.110.217.121 port 59810 connected with 128.110.217.160 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  10.8 GBytes  9.28 Gbits/sec
[  3]  0.0-10.0 sec  10.8 GBytes  9.28 Gbits/sec
```

System Performance Evaluation

Experiment Setup

In this section, we test the system's performance for three workloads: read-only, write-only, and 50% reads and 50% writes, by measuring their throughputs and latencies. For each workload, we run five replicas and begin by using a single client. We increase the number of clients by a factor of two until the throughput no longer increases. For each run, we initialize the store with 1,000,000 key-value pairs and the clients perform a total of 100,000 operations evenly distributed among themselves.

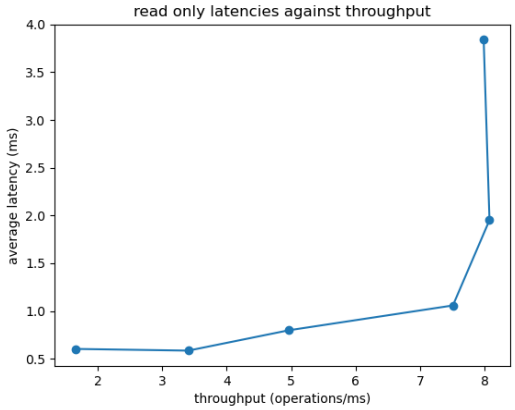
The server used in the testing is located in Utah.

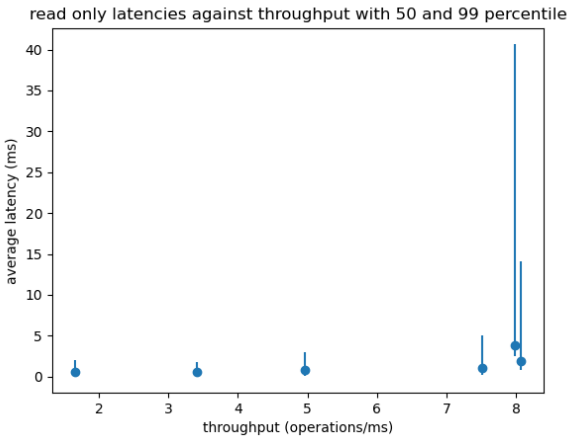
Hypothesis

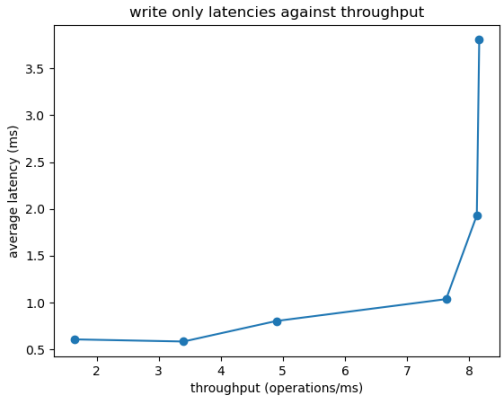
We hypothesize that as the number of clients increases, the throughput will initially increase as clients are initiated concurrently. Eventually, the throughput will reach a maximum value and remain stable as the replicas reach their maximum capability. The latency of each operation will initially increase slightly with the growing number of clients because the connections become busier. However, latency will then increase dramatically, especially when the throughput becomes stable, because traffic becomes even busier and more importantly, clients need to queue and wait a long time for the replicas to finish other clients' operations.

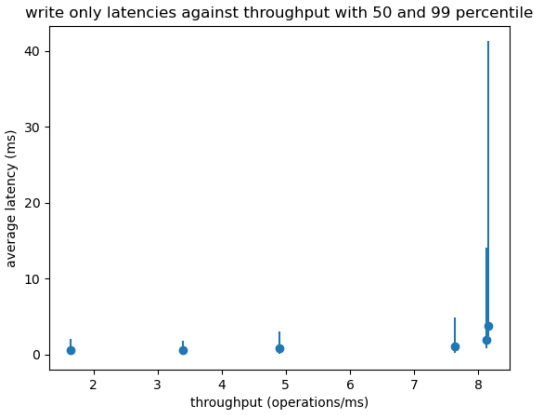
Results

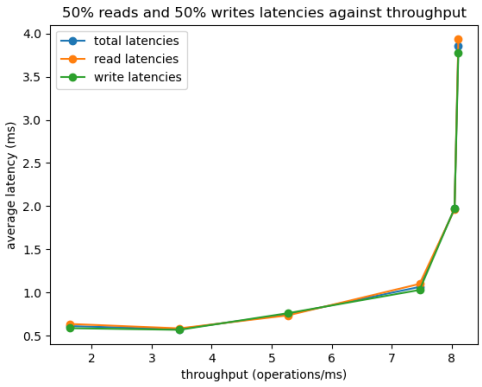
Below are plots of the average latency versus throughput for read-only, write-only, and 50% reads and 50% writes workloads. Additionally, for the 50% reads and 50% writes workload, we measure the average latency separately for reads and writes. For each workload, we also calculate the 50th and 99th percentile of the latencies and display plots of the latencies with error bars against the throughput. In addition, we measure the average, 50th, and 99th percentile latencies for the get and set operations separately for read and write. The results can be found in the log file.

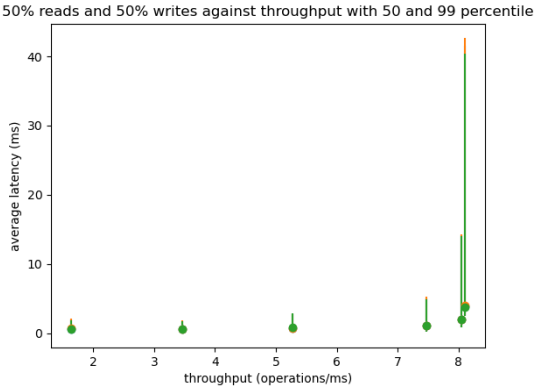












Discussion and Conclusions

The results basically confirm our hypothesis:

1. As more clients are added, the system experiences an initial increase in throughput due to concurrent client initiation and eventually reaches its maximum capability, resulting in a stable throughput.

2. Concurrently, latency increases slightly with the growing number of clients due to busier connections. Later, when throughput becomes stable, clients experience a significant increase in latency. This occurs because clients must wait in a queue for the replicas to finish other clients' operations, which becomes a bottleneck for the system.

We found that the median value of latencies is relatively smaller than the average values. However, the 99th percentile of latencies is significantly larger than the average values, and with an increase in clients, the difference between them becomes even greater. Based on this analysis, we can conclude that most operations are performed and responded to quickly by the replicas. However, due to the limited capacity of replicas, some operations may experience significant delays and remain queued for a long time, resulting in high latency.

Competitive Test with Raft

Experiment Setup

In this experiment, we compared our system's performance with etcd, a famous distributed key-value store based on Raft, under abnormal conditions. The setup and testing procedure for our system was generally identical to that of the previous experiment. For etcd, we also ran five servers on five physical machines and used the benchmark tool provided by etcd to obtain the results. After setting up the servers, we tested etcd as follows:

1. Detect the leader in our etcd cluster:
   ```
   etcdctl --endpoints=${ENDPOINTS} -w table endpoint status
   ```
2. Slow down the leader by limiting CPU usage:
   ```
   cpulimit -l ${percentage} -p ${pid} -b
   ```
   a. Or, fail the leader by killing the process: `pkill etcd`
3. Benchmark the etcd client by perform 100k r/w operations:
   ```
   benchmark --endpoints=${ENDPOINTS} --clients=100 txn-mixed
   --key-size=24 --total=100000 --val-size=10 --rw-ratio=1
   --consistency=l
   ```

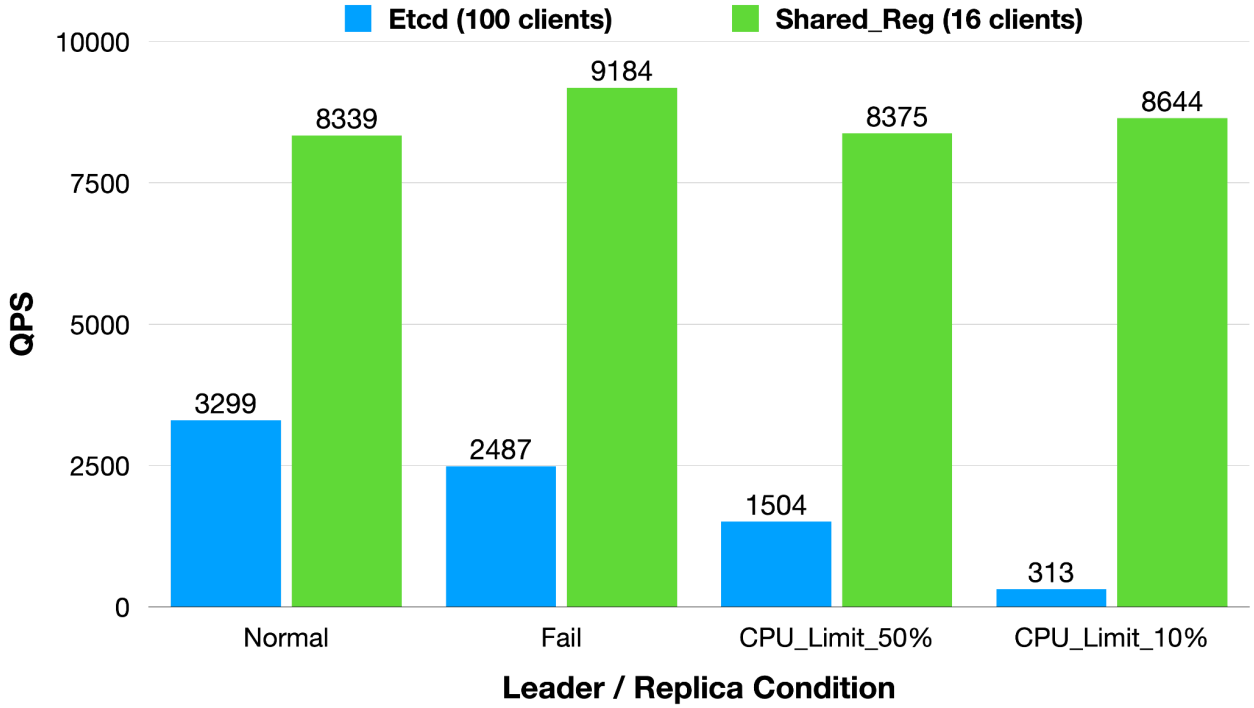
Hypothesis

We hypothesize that:

1. If the leader fails in etcd, it will automatically elect a new leader and the QPS (queries per second) will only experience a slight drop.
2. If the leader is slowed down in etcd, the QPS will drop significantly.
3. For the MWMR shared registers protocol, since a majority of the replicas are still running normally, if our concurrency design and timeout mechanism are reasonable, the QPS will only experience a slight drop.

Result

Below is the graph of the QPS for etcd and our system under four different experimental conditions. The performance of etcd conformed to our expectations, with a significant drop in QPS occurring when the CPU usage of the leader was severely limited. Surprisingly, the MWMR shared registers protocol performed even better when a replica was in trouble. We will discuss this phenomenon further in the next section.



Discussion and Conclusions

The result basically confirm to our hypothesis:

1. The performance of an etcd cluster can significantly drop when the leader is bottlenecked. This is because the leader is responsible for processing client requests, replicating state changes to other nodes, and ensuring that all nodes agree on the order of state changes. However, thanks to the re-election policy, a new leader is immediately elected when the leader fails, preventing a significant performance drop.
2. In contrast, for the MWMR shared registers protocol, when one of the replicas experiences issues, the performance may actually improve. This is likely due to the proper control of the behavior of each goroutine that sends gRPC requests, and the fact that the protocol only requires responses from a majority of the replicas. Therefore, the failure to receive responses from one replica will not significantly impact performance. Additionally, the client only needs to deal with less data, which increases the overall QPS.