Project 1: Replicated Storage and Consistency

CS 722: Cloud Computing Systems
10/29/2025

Samuel Harris & Kyle Goodwin

We built upon the actor system we had set up for Project 0 for the implementation of Project 1. This allowed us to have a consistent setup for each component (primary, backup1, backup2) that can send messages to each other asynchronously. Using ProtoActor was also helpful because it helped eliminate race conditions because of the built in locking/sequential message passing and parsing mechanisms. We didn't need to use Actors for Project 0 (as it called for traditional RPC, and was mentioned in the feedback), but we felt strongly that it would make our lives easier moving forward to Project 1 and eventually Project 2.

Our actors communicate using REST API style operations (GET, POST, READ, etc.) and we built our project around the fact that each VM can be either a primary or a backup, chosen by command line flags. Every operation on the primary receives an increasing LSN, which ensures ordering of operations that helps to prevent and track errors. The read on primary carries out a full replication process for linearizability and on backups reads are carried out locally to achieve better performance.

One of our main design decisions was to use a bit more tracking for the LSN (including lastAppliedLSN counter and methods like pendingCommits for a queue, etc.). We tried to lay out the actor code to be easy to follow and debug if errors occur or operations begin to happen out of order/become less strongly consistent.

Benchmarking

- CSV files can be found under 722_1P/benchmark/latency_logs/
- Testing was done over 4 Azure VMs on a virtual network. 3 VMs were set up running linux-distributed.exe which was the compiled code for actor.go, server.go, and main.go. These 3 VMs had state machines (actors) that connected to each other via RPC. The 4th VM was also on the virtual network but only communicated with the other 3 via HTTPs requests to the servers being run on each machine. (See 722_1P/README.md for more information on set up and running)

One client testing:

The default for one client testing was 10 bytes for values with a 60 second runtime. We'll include the command we run and the results from each test:

```
./linux-benchmark.exe -primary=10.0.0.4:8081 -backup1=10.0.0.5:8083 -
backup2=10.0.0.6:8085 -rwratio=0.5 -readfromlog=0 -duration=60
```

Run 1:
Total operations: 48233
Duration: 60.10 seconds
Throughput: 801.54 operations/sec
Average latency: 0.7066 ms
99th percentile: 2.0000 ms
99.999th percentile: 50.3885 ms
Run 2:
Total Operations: 50842
Duration: 60.10 seconds
Throughput: 845.95 operations/sec
Average latency: 0.7066 ms
99th percentile: 2.0000 ms
99.999th percentile: 50.3885 ms
Run 3:
Total Operations: 49691
Duration: 60.10 seconds
Throughput: 826.80 operations/second
Average latency: 0.6652 ms
99th percentile: 2.0000 ms
99.999th percentile: 30.5462 ms
Analysis:

The average throughput across the three tests came to 824.73 operations/second. There weren't huge variations between the numbers in each of the tests, they were quite consistent. The CSV files for these tests reveal a cold start, with the first few operations taking far longer (around 30ms) and the operations after that dropping down to a consistent 1, 2, or 3ms.

Two client testing:

The default for two client testing was 10 bytes for values with a 60 second runtime. We'll include the command we run and the results from each test:

```
./linux-benchmark.exe -primary=10.0.0.4:8081 -backup1=10.0.0.5:8083 -
backup2=10.0.0.6:8085 -rwratio=0.5 -readfromlog=0 -duration=60 -clients=2
```

Run 1:
Total operations: 90124
Duration: 60.10 Seconds
Throughput: 1499.55 ops/sec
Average latency: 0.8668 ms
99th percentile: 2.0000 ms
99.999th percentile: 22.0987 ms
Run 2:
Total operations:70752
Duration: 60.10 Seconds
Throughput: 1177.23 ops/sec
Average latency: 1.2031 ms
99th percentile: 2.0000 ms
99.999th percentile: 38768.2506 ms
Note: Noticeable jitters in the print log for a couple seconds
Run 3:
Total operations: 60435
Duration: 75.95 Seconds
Throughput: 795.73 ops/sec
Average latency: 1.7400 ms
99th percentile: 2.0000 ms
99.999th percentile: 30011.1651 ms
Note: Significant pause near the end of duration and Request timeout error logged in primary
Run 4:
Total operations: 90691
Duration: 60.10 Seconds
Throughput: 1508.98 ops/sec
Average latency: 1.7400 ms
99th percentile: 2.0000 ms
99.999th percentile: 30011.1651 ms

Analysis:

The average throughput for 2 clients was 1245.37 ops/sec. This is just a 15% increase in throughput over 1 client. Although if you take the low outlier of run 3 the average is around 1400 ops/sec. Latency increased by around ½ to 1 ms over the 1 client benchmarking. Still with high cold-start latency around 30ms for each run.

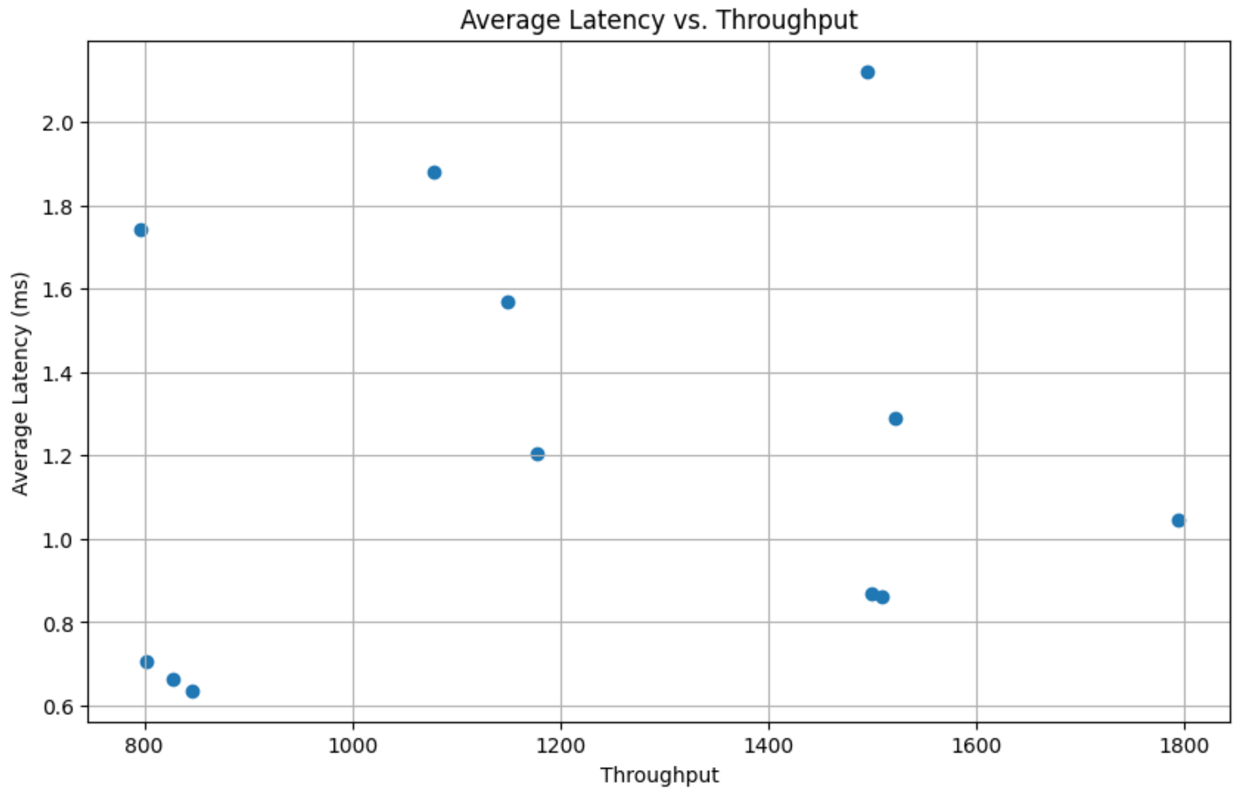
Three client testing:

```
./linux-benchmark.exe -primary=10.0.0.4:8081 -backup1=10.0.0.5:8083 -
backup2=10.0.0.6:8085 -rwratio=0.5 -readfromlog=0 -duration=60 -clients=3
```

Run 1
Total operations: 101357
Duration: 88.24 Seconds
Throughput: 1148.64 ops/sec
Note: Pause at end of duration and request timeout logged in primary
Average latency: 0.8614 ms
99th percentile: 2.0000 ms
99.999th percentile: 23.0931 ms
Run 2
Total operations: 95982
Duration: 89.02
Throughput: 1078.23 ops/sec
Average latency: 1.8802 ms
99th percentile: 2.0000 ms
Run 3 NOTE: Fixed abrupt ending causing lag and low ops
Total operations: 116822
Duration: 65.10 Seconds
Throughput: 1794.45 ops/sec
Average latency: 1.0468 ms
99th percentile: 2.0000 ms
99.999th percentile: 25.0000 ms
Run 4
Total operations: 103066
Duration: 67.70 seconds
Throughput: 1522.33 ops/sec
Average latency: 1.2883 ms
99th percentile: 3.0000 ms
99.999th percentile: 29081.6361 m
Run 5: 5 Clients
Total operations: 121939
Duration: 81.54
Throughput: 1495.44 ops/sec
Average latency: 2.1210 ms
99th percentile: 3.0000 ms
99.999th percentile: 30001.0000 ms

Analysis:

The throughput increased with more client, however with more than 2 clients a problem was occurring where there was lag when the test ended. we believe that it was cutting off in-flight requests and to abruptly stopping. So, when we added a simple wait, there was much higher throughput and total operations. As well as an increase in latency although, in the struggling runs the latency reduced as throughput did. We don't see the expected curve of linear connection between latency and throughput. We think the reason for this is some unexpected behavior from weird lag and the problem of abrupt benchmark stopping before that was fixed.



Multiple clients, Read from backups testing

```
./linux-benchmark.exe -primary=10.0.0.4:8081 -backup1=10.0.0.5:8083 -
backup2=10.0.0.6:8085 -rwratio=0.5 -readfromlog=1 -duration=60 -clients=2
```

Run 1:
Total Operations: 130167
Duration: 65.10 seconds
Throughput: 1999.46 ops/sec
Average Latency: 0.47 ms
Run 2:
Total operations: 132147
Duration: 65.10 seconds
Throughput: 2029.86 ops/sec
Average Latency .46 ms
Analysis

By moving reads to the backup's throughput increases significantly while latency decreases similarly. We couldn't get a stale read to occur. However, it is possible. For that to occur we would have to have multiple writes for the same key. The first write can be stored correctly by all machines. Then the second write might reach only the primary and 1 backup while the other backup is working on other requests (maybe it had a lot of reads).

Then with a quorum, that second write is committed. But before the lagging backup commits it to a store, it has a read for the same key and serves the stale initial write. Even though a new write has been committed.

AI Usage

Similarly to our process in Project 0, we laid out what we thought would be a good "baseline" for our expansion into Project 1, which included adding maps to each actor to store actions and the layout for our REST API style communication. Most of main.go and actors.go was brough over from Project 0. Although the actor struct had to be adjusted as we added different functionality but that was easily done without AI. What we used AI for primarily was in actors.go when we were working through how to use a queue sort of structure to keep track of what LSNs could actually be committed to the store. We knew we wanted to have a queue that keeps track of what requests/LSNs need to be committed to the store but couldn't because they are waiting on LSN-1. So, the idea was whenever a LSN is committed to store, that thread goes to the queue and starts a chain reaction to commit old values that have been waiting. We mainly used Anthropic Claude 3.5-4.5 to converse with and generate both code and ideas for this expansion and implementation. Claude was particularly helpful with some of the LSN logic (functions like applyLSNToBackup, applyLSNToPrimary, among other functions in actors.go). In addition, it wrote a majority of the *test_commands.sh* script that was super helpful in the beginning stages of the project to make sure our changes worked as we expected.

We took the feedback from Project 0 (trying not to prompt AI with the "bigger questions) and we think it helped a lot with trying to generate and target specific issues rather than just having it do most of the work for us. Writing our own comments and prompting Claude to do so also helped when debugging and making changes so it was more obvious what each piece of code was doing.

When creating Benchmark.go We had originally written out the benchmark logic on my own. The code for running the benchmark and determining what operations, how many operations, and how many clients were done. However, we realized that my solution was not running the clients concurrently and clients had to wait for each other in a for loop before sending more requests. So, we prompted Claude Sonnet 4.5 via GitHub Copilot to modify my Benchmark.go "Using channels, there should always be client's number of clients running concurrently making requests. This section of the code should record the time before starting a client, and have the client make a request. When the client is done it puts its result in a channel which the main thread has been waiting for and then gets the current time and calculates the RTT for that operation which is logged in the csv file". The edits made changed what we had already to start clients concurrently using a waitgroup as well as saving logging in memory before writing to a csv. Most of my code was worked in or we moved it to getURL which determined what operation to send. we checked over this code by running the benchmark locally and ensuring the logs made sense. As well as comparing it to the goroutine use we had implemented in project 0. There was one issue that Claude messed up and was only making http. Get requests so we made a simple change to ensure writes used post and reads used get.