**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 reads 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.

**Testing Results**

* CSV files can be found under 722\_1P/benchmark/latency\_logs/
* Our test driver used randomized keys for writes and the reads chose a random key from the list of stored keys that have already been written.
* These tests were done locally (can be seen in the command used to run the test) – this is the reason for fast times and high throughput

**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

**Run 2:**

Total Operations: 50842

Duration: 60.10 seconds

Throughput: 845.95 operations/sec

**Run 3:**

Total Operations: 49691

Duration: 60.10 seconds

Throughput: 826.80 operations/second

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

**Run 2:**

Total operations:70752

Duration: 60.10 Seconds

Throughput: 1177.23 ops/sec

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

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

**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

Run 2

Total operations: 95982

Duration: 89.02

Throughput: 1078.23 ops/sec

Run 3 NOTE: Fixed abrupt ending causing lag and low ops

Total operations: 116822

Duration: 65.10 Seconds

Throughput: 1794.45 ops/sec

Run 4

Total operations: 103066

Duration: 67.70 seconds

Throughput: 1522.33 ops/sec

Run 5: 5 Clients

Total operations: 121939

Duration: 81.54

Throughput: 1495.44 ops/sec

**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

Run 2:

Total operations: 132147

Duration: 65.10 seconds

Throughput: 2029.86 ops/sec

**Performance Analysis**

- Single client baseline: 824.73 ops/sec average

- Scaling efficiency: 2 clients = 1.8x, 3 clients = 1.9x throughput

- Read optimization: Reading from backups improved throughput by ~50%

- Bottleneck: Primary appears to be the bottleneck (timeout errors with 3+ clients)

Known Issues

- Timeout errors occur with 3+ concurrent clients

- Cold start penalty: ~30ms for first operations

**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. 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 a majority 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 actually doing.