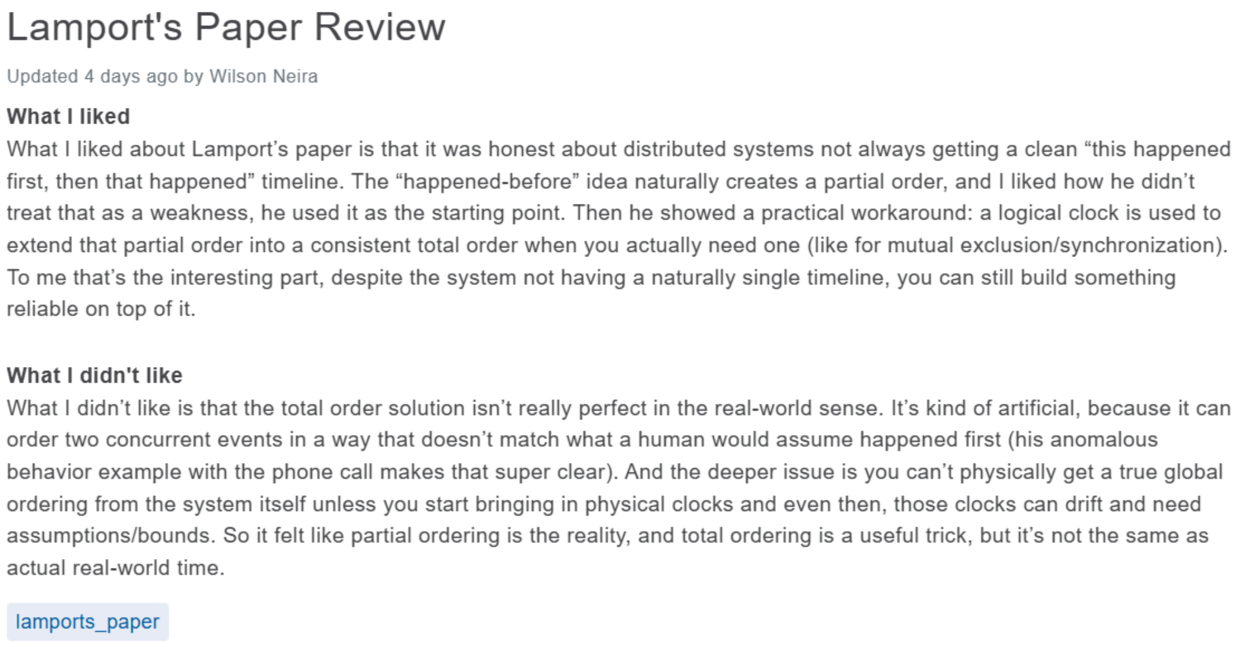
Homework 3 Report
Wilson Neira

Part I: Time and Ordering!



Lamport's Paper Review

Updated 4 days ago by Wilson Neira

What I liked

What I liked about Lamport's paper is that it was honest about distributed systems not always getting a clean "this happened first, then that happened" timeline. The "happened-before" idea naturally creates a partial order, and I liked how he didn't treat that as a weakness, he used it as the starting point. Then he showed a practical workaround: a logical clock is used to extend that partial order into a consistent total order when you actually need one (like for mutual exclusion/synchronization). To me that's the interesting part, despite the system not having a naturally single timeline, you can still build something reliable on top of it.

What I didn't like

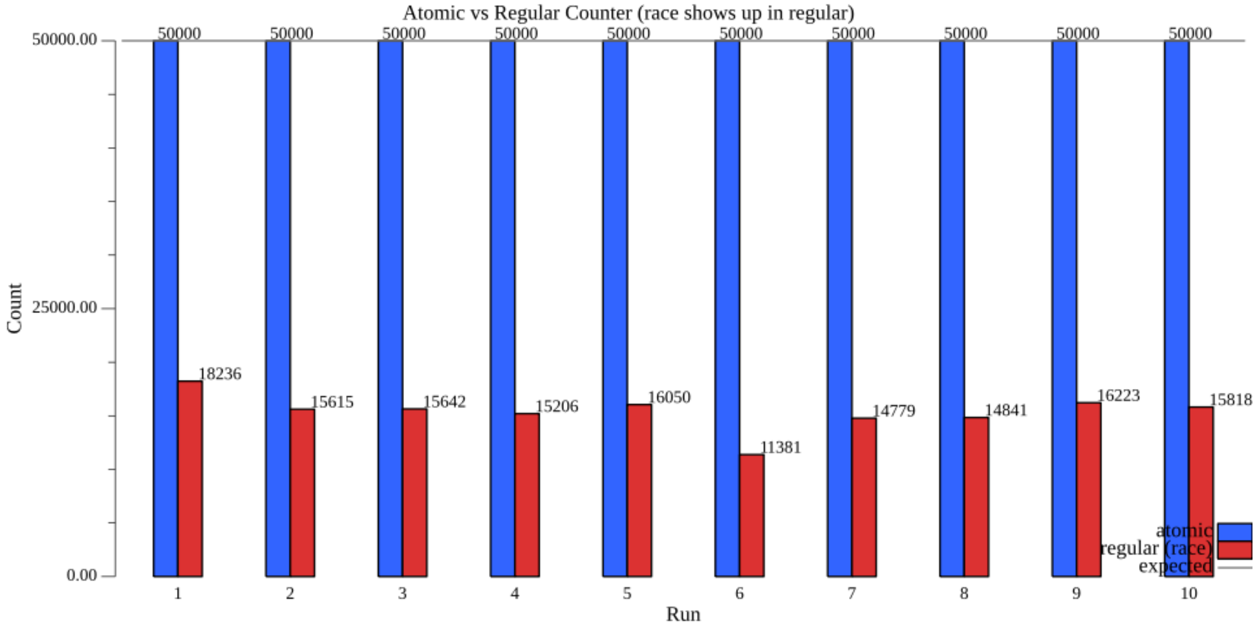
What I didn't like is that the total order solution isn't really perfect in the real-world sense. It's kind of artificial, because it can order two concurrent events in a way that doesn't match what a human would assume happened first (his anomalous behavior example with the phone call makes that super clear). And the deeper issue is you can't physically get a true global ordering from the system itself unless you start bringing in physical clocks and even then, those clocks can drift and need assumptions/bounds. So it felt like partial ordering is the reality, and total ordering is a useful trick, but it's not the same as actual real-world time.

lamports_paper

Part II: Thread Experiments Overview

Atomicity

Run the exercise on your system and compare an atomic integer vs regular integer. Note, you may have to run it several times to see the race condition happen!

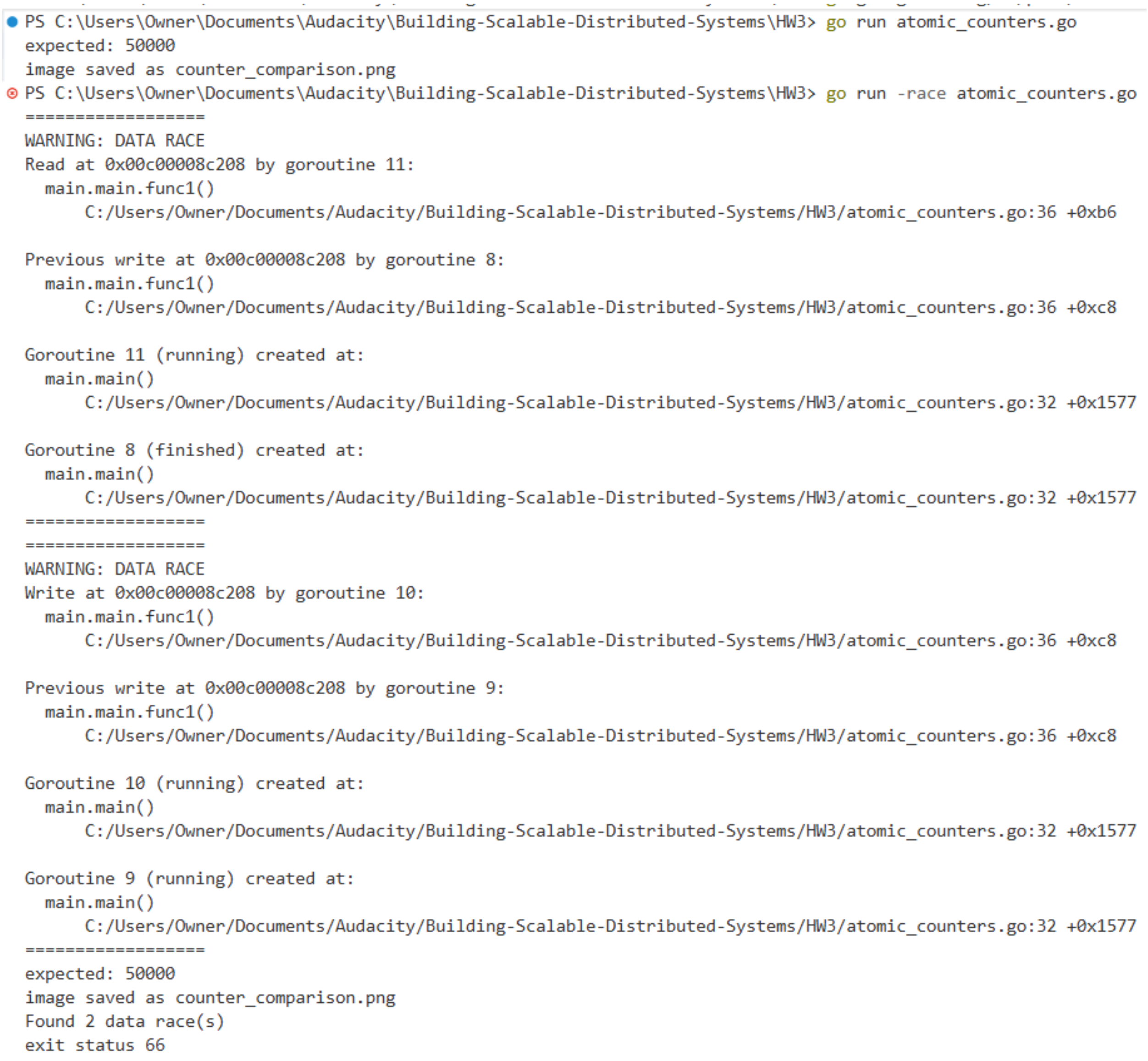


What values do you see?

I see a constant 50000 count for the atomic integer. While I see varying results like 18236, 15615, and 15642 for the regular integer. When the expected value is 50000 for atomic and regular integers.

What is happening?
What's happening is the atomic counter is thread-safe, so it always reaches the correct 50000, but the regular counter has a race condition where multiple goroutines overwrite each other's increments, so it loses updates and ends up with random lower numbers each run.

Try running with the -race flag. What does that do?
Running with the '-race' flag turns on Go's race detector, which actively checks for unsafe concurrent memory access and clearly reports that the regular counter has data races while the atomic counter does not.



```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run atomic_counters.go
expected: 50000
image saved as counter_comparison.png
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run -race atomic_counters.go
==================
WARNING: DATA RACE
Read at 0x00c00008c208 by goroutine 11:
  main.main.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:36 +0xb6

Previous write at 0x00c00008c208 by goroutine 8:
  main.main.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:36 +0xc8

Goroutine 11 (running) created at:
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:32 +0x1577

Goroutine 8 (finished) created at:
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:32 +0x1577
==================
==================
WARNING: DATA RACE
Write at 0x00c00008c208 by goroutine 10:
  main.main.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:36 +0xc8

Previous write at 0x00c00008c208 by goroutine 9:
  main.main.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:36 +0xc8

Goroutine 10 (running) created at:
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:32 +0x1577

Goroutine 9 (running) created at:
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/atomic_counters.go:32 +0x1577
==================
expected: 50000
image saved as counter_comparison.png
Found 2 data race(s)
exit status 66
```

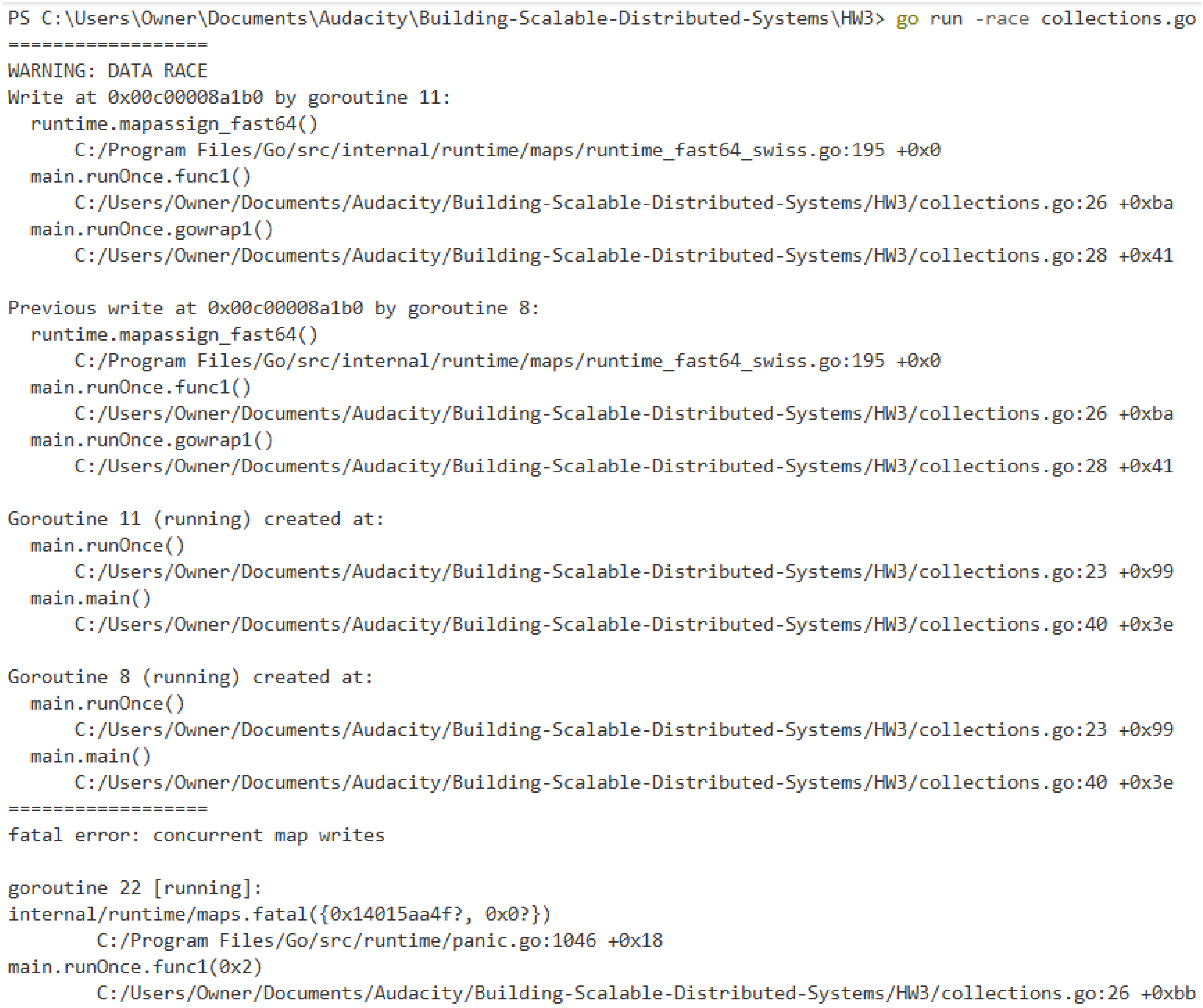
Collections

You might see that your program crashed. Why?

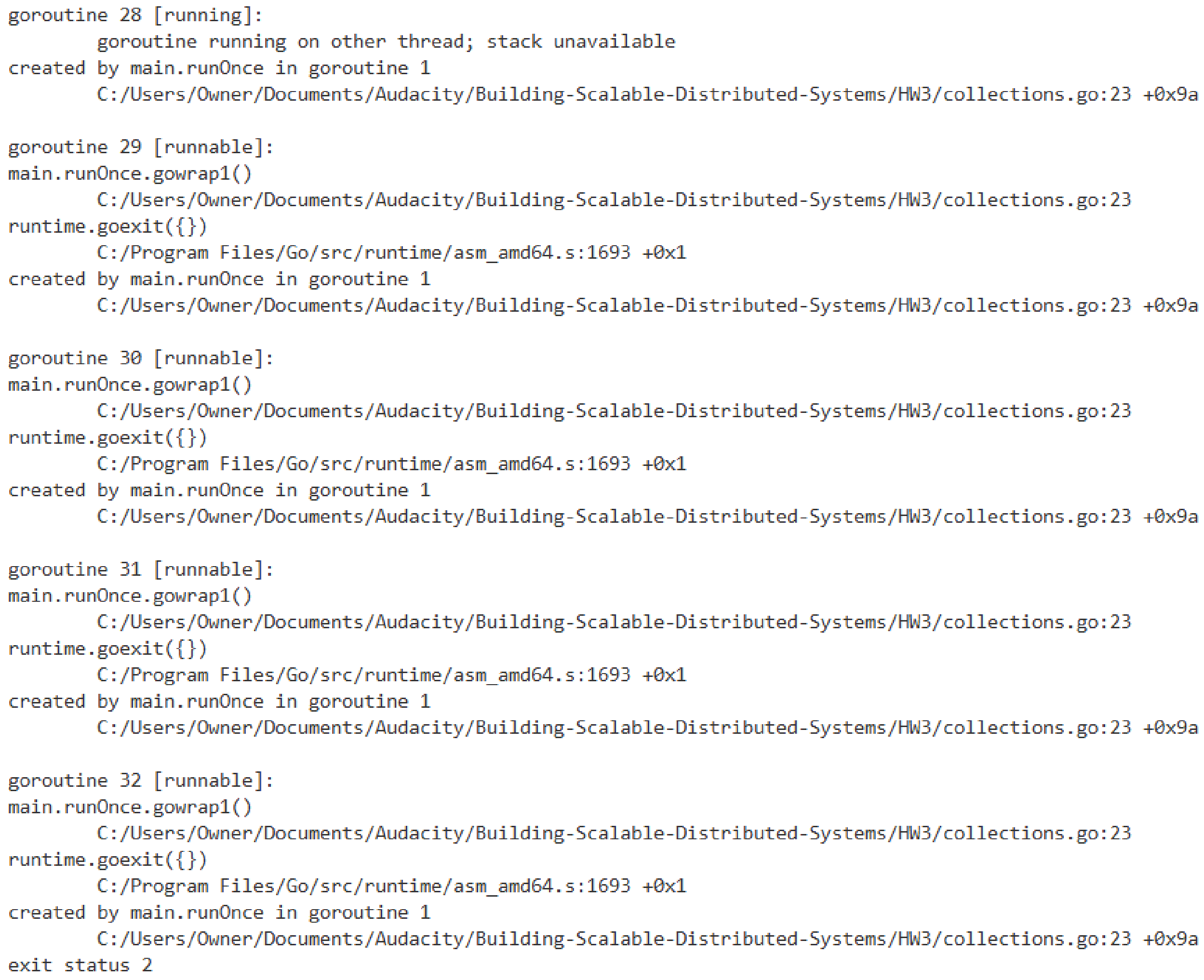
Because a plain Go map isn't safe to write to from multiple goroutines at the same time, so Go panics with “concurrent map writes.”

What's going on?

The 50 goroutines are all trying to do m[...] = … at once, and 2 of them hit the map at the same time (the race report shows that), which corrupts the map's internal state and triggers the fatal error.

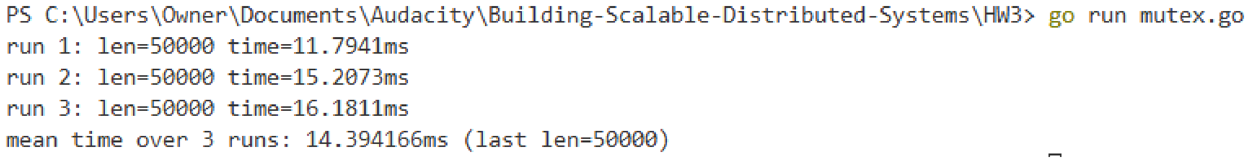


```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run -race collections.go
==================
WARNING: DATA RACE
Write at 0x00c00008a1b0 by goroutine 11:
  runtime.mapassign_fast64()
      C:/Program Files/Go/src/internal/runtime/maps/runtime_fast64_swiss.go:195 +0x0
  main.runOnce.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:26 +0xba
  main.runOnce.gowrap1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:28 +0x41

Previous write at 0x00c00008a1b0 by goroutine 8:
  runtime.mapassign_fast64()
      C:/Program Files/Go/src/internal/runtime/maps/runtime_fast64_swiss.go:195 +0x0
  main.runOnce.func1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:26 +0xba
  main.runOnce.gowrap1()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:28 +0x41

Goroutine 11 (running) created at:
  main.runOnce()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x99
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:40 +0x3e

Goroutine 8 (running) created at:
  main.runOnce()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x99
  main.main()
      C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:40 +0x3e
==================
fatal error: concurrent map writes

goroutine 22 [running]:
internal/runtime/maps.fatal({0x14015aa4f?, 0x0?})
        C:/Program Files/Go/src/runtime/panic.go:1046 +0x18
main.runOnce.func1(0x2)
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:26 +0xbb
```



```
goroutine 28 [running]:
        goroutine running on other thread; stack unavailable
created by main.runOnce in goroutine 1
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x9a

goroutine 29 [runnable]:
main.runOnce.gowrap1()
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23
runtime.goexit({})
        C:/Program Files/Go/src/runtime/asm_amd64.s:1693 +0x1
created by main.runOnce in goroutine 1
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x9a

goroutine 30 [runnable]:
main.runOnce.gowrap1()
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23
runtime.goexit({})
        C:/Program Files/Go/src/runtime/asm_amd64.s:1693 +0x1
created by main.runOnce in goroutine 1
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x9a

goroutine 31 [runnable]:
main.runOnce.gowrap1()
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23
runtime.goexit({})
        C:/Program Files/Go/src/runtime/asm_amd64.s:1693 +0x1
created by main.runOnce in goroutine 1
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x9a

goroutine 32 [runnable]:
main.runOnce.gowrap1()
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23
runtime.goexit({})
        C:/Program Files/Go/src/runtime/asm_amd64.s:1693 +0x1
created by main.runOnce in goroutine 1
        C:/Users/Owner/Documents/Audacity/Building-Scalable-Distributed-Systems/HW3/collections.go:23 +0x9a
exit status 2
```

Mutex

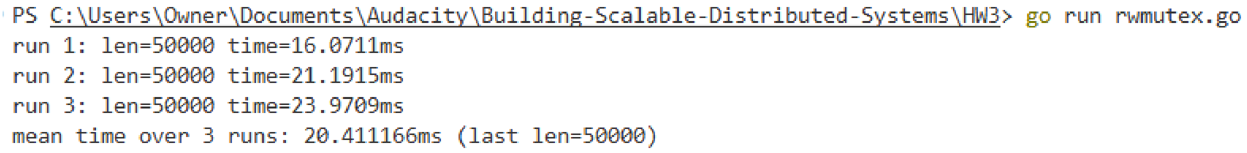


```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run mutex.go
run 1: len=50000 time=11.7941ms
run 2: len=50000 time=15.2073ms
run 3: len=50000 time=16.1811ms
mean time over 3 runs: 14.394166ms (last len=50000)
```

What's the lesson learned here?

The lesson is that a plain Go map will break or give wrong results when lots of goroutines write at once, so you must synchronize it (like with a mutex) to make it safe and consistent.

RWMutex



```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run rwmutex.go
run 1: len=50000 time=16.0711ms
run 2: len=50000 time=21.1915ms
run 3: len=50000 time=23.9709ms
mean time over 3 runs: 20.411166ms (last len=50000)
```

Did this change anything?

Yeah, the RWMutex version was actually a bit slower than the Mutex version (about ~20.4ms vs ~14.4ms average).

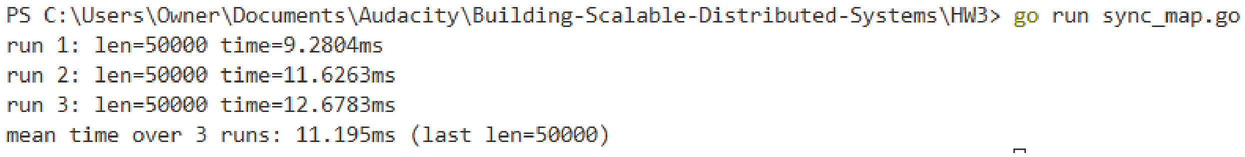
Why or why not?

Because the workload is basically all writes, so RWMutex doesn't get its many readers at once advantage like before and just adds extra overhead.

What is the lesson learned here?

RWMutex likely only helps when you have lots of reads and few writes, if it's mostly writes, a normal Mutex is usually just as good or better.

Sync.Map

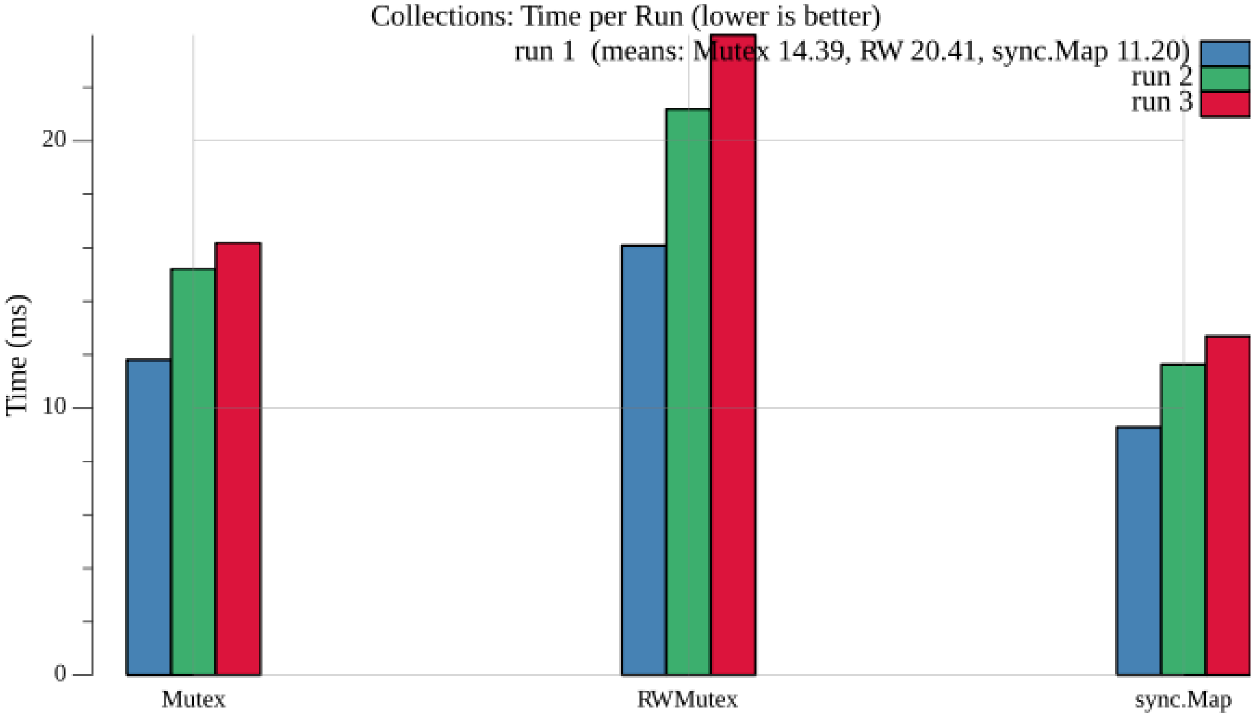


```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run sync_map.go
run 1: len=50000 time=9.2804ms
run 2: len=50000 time=11.6263ms
run 3: len=50000 time=12.6783ms
mean time over 3 runs: 11.195ms (last len=50000)
```

What is the lesson learned now?

sync.Map handled the concurrent writes safely and faster than locking a normal map, so the right data structure can beat adding a mutex everywhere.

Make sure that you can summarize the tradeoffs in the different approaches above! More specifically, directly compare 3 sets of results you obtained for each tests above in some quantitative representation (this can be creative!).

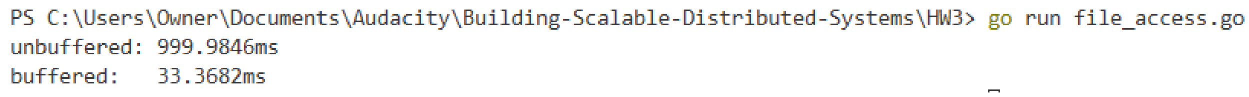


Explain the reasons behind these results.

Because this workload is mostly writes, and some approaches add more locking overhead than others.

In this case, we are mostly writing to the map, but what will happen if read operations dominate?
If there are lots of reads and few writes, RWMutex or sync.Map can do better because many readers can run at the same time.
You might have observed that one of these is definitely the fastest, but what are the tradeoffs between these approaches?
sync.Map is the fastest and is optimized for concurrent access and avoids heavy lock contention. In terms of tradeoffs a plain map with mutex is simple and type-safe, RWMutex only helps with read-heavy workloads, and sync.Map is fastest for concurrency but is less type-safe and more specialized.

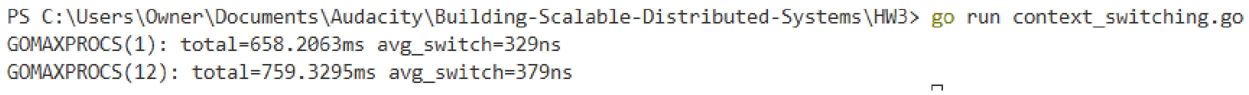
File Access



```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run file_access.go
unbuffered: 999.9846ms
buffered:   33.3682ms
```

What?
Buffered writing was way faster (about 33ms vs ~1000ms, ~30× faster).
Why?
Unbuffered does a real OS write basically every line (tons of expensive syscalls), while buffered batches many lines in memory and writes them out in big chunks.
What is going on and what are the lessons learned about tradeoffs from these results?
The bottleneck isn’t writing text, it’s the overhead of repeatedly crossing into the OS/disk layer thousands of times. Buffering gives huge speedups, but you must flush (or you lose recent data on crash), it uses some extra memory, and it can delay when data hits disk.

Context Switching

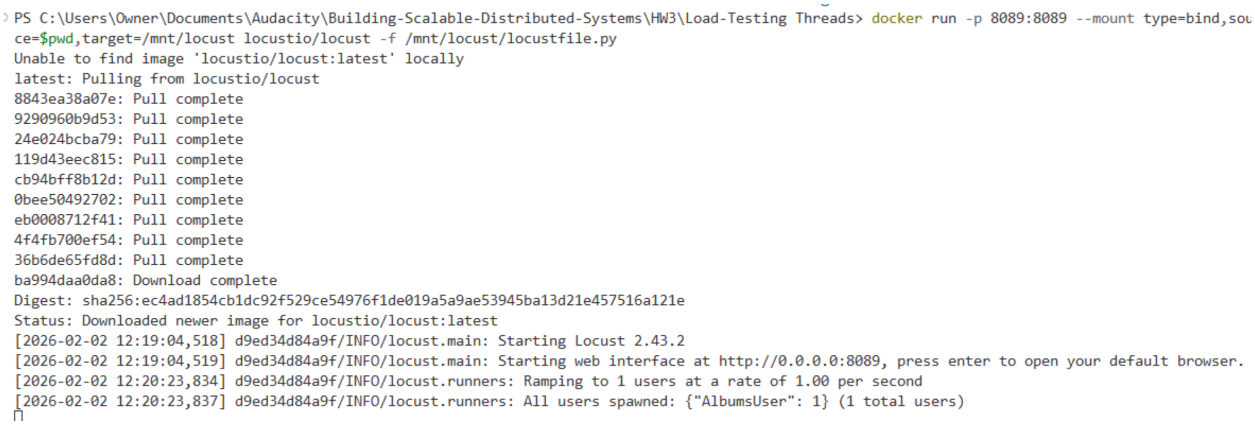


```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3> go run context_switching.go
GOMAXPROCS(1): total=658.2063ms avg_switch=329ns
GOMAXPROCS(12): total=759.3295ms avg_switch=379ns
```

Which one is faster?
GOMAXPROCS(1) is faster here (329ns vs 379ns).
Why?
With 1 OS thread the goroutines handoff without bouncing between CPU threads/cores, so there’s less scheduling/coordination overhead.
How do you think this relates to the cost of context switching between processes, containers, and virtual machines?
Goroutine switches are tiny (nanoseconds), but switching between processes is heavier, containers are about like processes (container != VM), and VM context switches are usually the most expensive because there’s more isolation/state to manage.

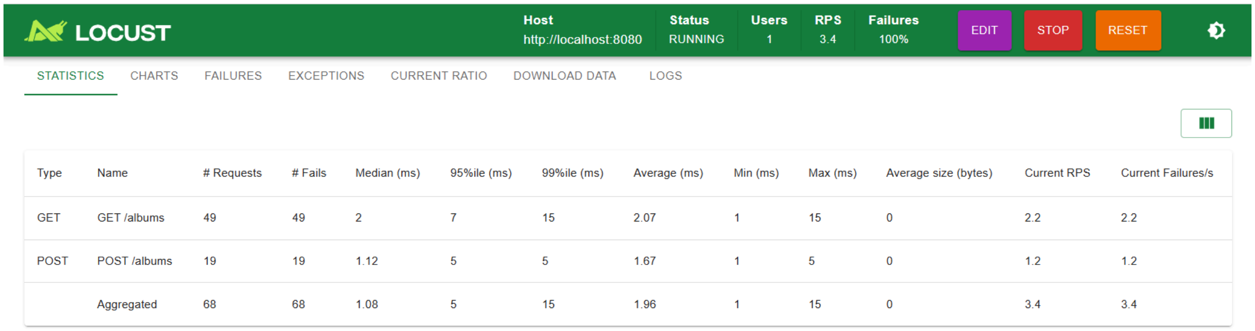
Part III: Making Threads work hard with Load-Testing!

Locust

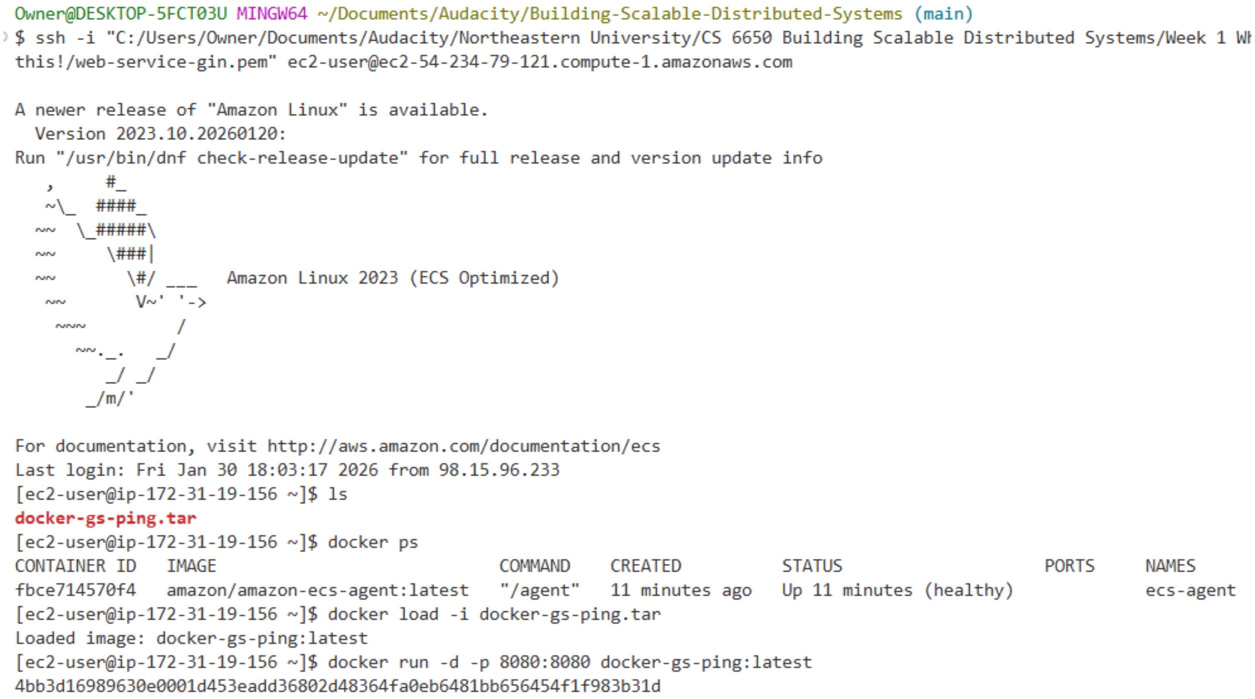


```
PS C:\Users\Owner\Documents\Audacity\Building-Scalable-Distributed-Systems\HW3\Load-Testing Threads> docker run -p 8089:8089 --mount type=bind,sou
ce=$pwd,target=/mnt/locust locustio/locust -f /mnt/locust/locustfile.py
Unable to find image 'locustio/locust:latest' locally
latest: Pulling from locustio/locust
8843ea38a07e: Pull complete
9290960b9d53: Pull complete
24e024bcba79: Pull complete
119d43eec815: Pull complete
cb94bff8b12d: Pull complete
0bee50492702: Pull complete
eb0008712f41: Pull complete
4f4fb700ef54: Pull complete
36b6de65fd8d: Pull complete
ba994daa0da8: Download complete
Digest: sha256:ec4ad1854cb1dc92f529ce54976f1de019a5a9ae53945ba13d21e457516a121e
Status: Downloaded newer image for locustio/locust:latest
[2026-02-02 12:19:04,518] d9ed34d84a9f/INFO/locust.main: Starting Locust 2.43.2
[2026-02-02 12:19:04,519] d9ed34d84a9f/INFO/locust.main: Starting web interface at http://0.0.0.0:8089, press enter to open your default browser.
[2026-02-02 12:20:23,834] d9ed34d84a9f/INFO/locust.runners: Ramping to 1 users at a rate of 1.00 per second
[2026-02-02 12:20:23,837] d9ed34d84a9f/INFO/locust.runners: All users spawned: {"AlbumsUser": 1} (1 total users)
```

Before Running EC2

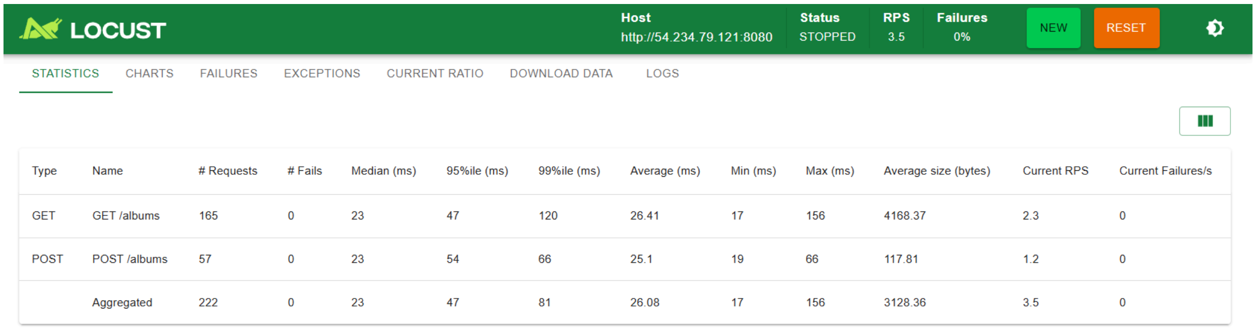


Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	49	49	2	7	15	2.07	1	15	0	2.2	2.2
POST	POST /albums	19	19	1.12	5	5	1.67	1	5	0	1.2	1.2
	Aggregated	68	68	1.08	5	15	1.96	1	15	0	3.4	3.4



```
Owner@DESKTOP-5FCT03U MINGW64 ~/Documents/Audacity/Building-Scalable-Distributed-Systems (main)
$ ssh -i "C:/Users/Owner/Documents/Audacity/Northeastern University/CS 6650 Building Scalable Distributed Systems/Week 1 Wh
this!/web-service-gin.pem" ec2-user@ec2-54-234-79-121.compute-1.amazonaws.com

A newer release of "Amazon Linux" is available.
  Version 2023.10.20260120:
Run "/usr/bin/dnf check-release-update" for full release and version update info
   ,     #_
   ~\_  ####_
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   Amazon Linux 2023 (ECS Optimized)
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'

For documentation, visit http://aws.amazon.com/documentation/ecs
Last login: Fri Jan 30 18:03:17 2026 from 98.15.96.233
[ec2-user@ip-172-31-19-156 ~]$ ls
docker-gs-ping.tar
[ec2-user@ip-172-31-19-156 ~]$ docker ps
CONTAINER ID   IMAGE                            COMMAND    CREATED          STATUS                    PORTS     NAMES
fbce714570f4   amazon/amazon-ecs-agent:latest   "/agent"   11 minutes ago   Up 11 minutes (healthy)             ecs-agent
[ec2-user@ip-172-31-19-156 ~]$ docker load -i docker-gs-ping.tar
Loaded image: docker-gs-ping:latest
[ec2-user@ip-172-31-19-156 ~]$ docker run -d -p 8080:8080 docker-gs-ping:latest
4bb3d16989630e0001d453eadd36802d48364fa0eb6481bb656454f1f983b31d
```

After Running EC2



Host http://54.234.79.121:8080 | Status STOPPED | RPS 3.5 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	165	0	23	47	120	26.41	17	156	4168.37	2.3	0
POST	POST /albums	57	0	23	54	66	25.1	19	66	117.81	1.2	0
	Aggregated	222	0	23	47	81	26.08	17	156	3128.36	3.5	0

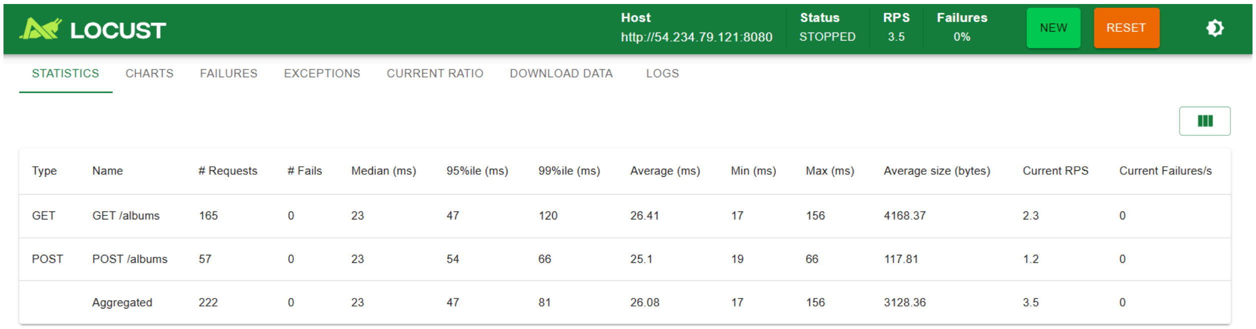
Which operations will be most common in real world scenario?
In a real app, GET/read requests are usually way more common than POST/writes because most people are just viewing data most of the time.

How does that impact the data structure you are using to save your data?
If reads are most common, you likely want a data structure optimized for fast lookups/reads (like an in-memory map/hash table, maybe alongside a list for ordered output).

After you have both requests in place with no failures, capture screenshots showing the difference in statistics between GET and POST. What is going on here?
What's happening in my screenshot is both GET and POST are succeeding, and their latencies are pretty close (GET avg ~26ms, POST avg ~25ms).

Can you argue reasons behind different numbers?
The small differences (like GET having a higher max/99th) are probably because GET returns a bigger response and may loop/serialize more data, while POST sends a tiny body and just appends one item.

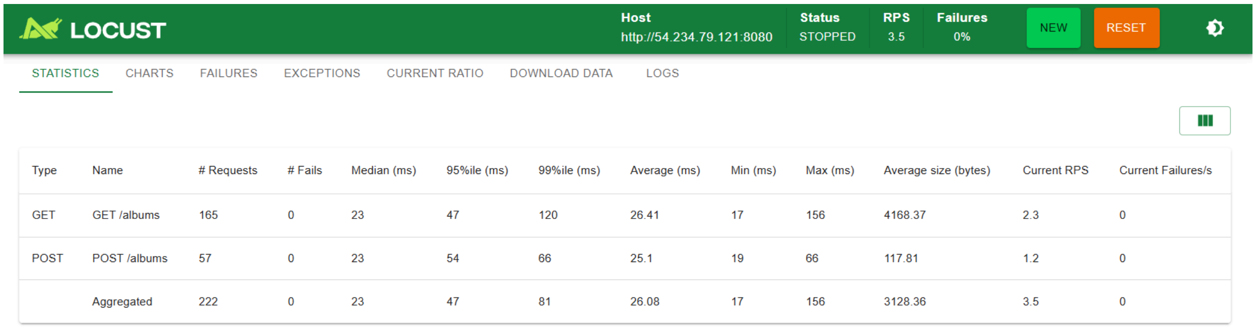
Local Test
GET and POST tasks ratios 3:1 (Code was already 3:1)



Host http://54.234.79.121:8080 | Status STOPPED | RPS 3.5 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	165	0	23	47	120	26.41	17	156	4168.37	2.3	0
POST	POST /albums	57	0	23	54	66	25.1	19	66	117.81	1.2	0
	Aggregated	222	0	23	47	81	26.08	17	156	3128.36	3.5	0

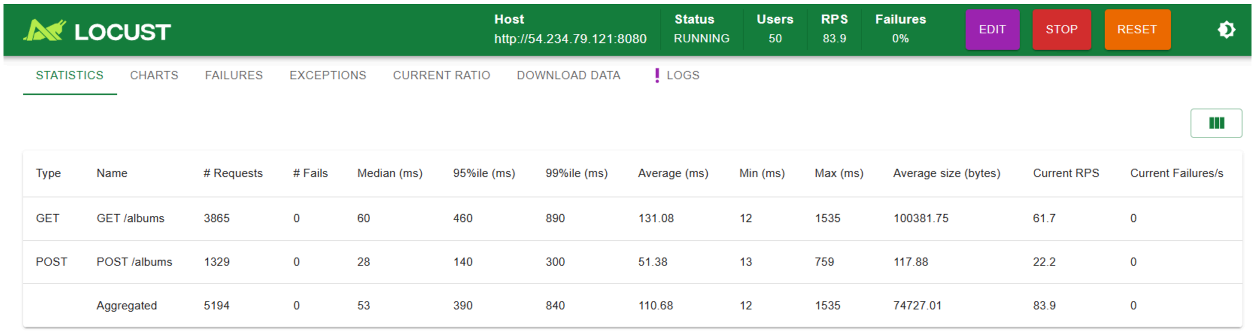
1 worker (I already ran for 1 Locus process generating load)



Host http://54.234.79.121:8080 | Status STOPPED | RPS 3.5 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	165	0	23	47	120	26.41	17	156	4168.37	2.3	0
POST	POST /albums	57	0	23	54	66	25.1	19	66	117.81	1.2	0
	Aggregated	222	0	23	47	81	26.08	17	156	3128.36	3.5	0

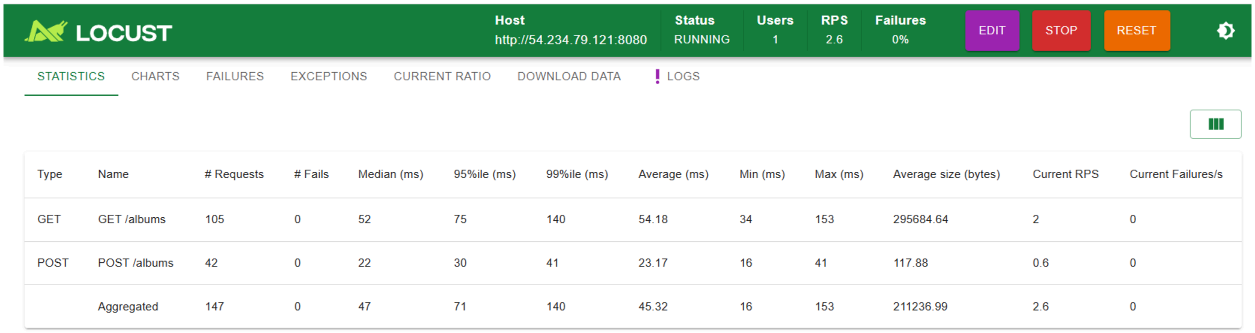
50 users



Host http://54.234.79.121:8080 | Status RUNNING | Users 50 | RPS 83.9 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	3865	0	60	460	890	131.08	12	1535	100381.75	61.7	0
POST	POST /albums	1329	0	28	140	300	51.38	13	759	117.88	22.2	0
	Aggregated	5194	0	53	390	840	110.68	12	1535	74727.01	83.9	0

10 users per second ramp up time

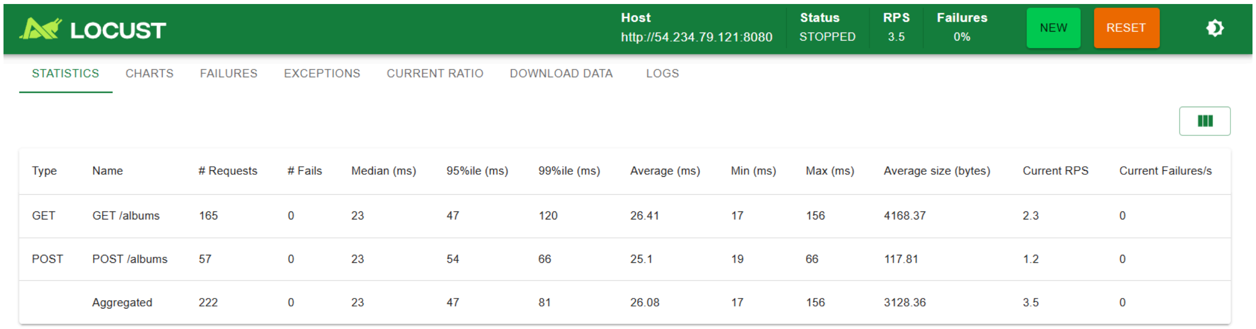


Host http://54.234.79.121:8080 | Status RUNNING | Users 1 | RPS 2.6 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	105	0	52	75	140	54.18	34	153	295684.64	2	0
POST	POST /albums	42	0	22	30	41	23.17	16	41	117.88	0.6	0
	Aggregated	147	0	47	71	140	45.32	16	153	211236.99	2.6	0

Amdahl's Law

Record the statistics of GET vs POST with 1 worker (already done before).



Host http://54.234.79.121:8080 | Status STOPPED | RPS 3.5 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	165	0	23	47	120	26.41	17	156	4168.37	2.3	0
POST	POST /albums	57	0	23	54	66	25.1	19	66	117.81	1.2	0
	Aggregated	222	0	23	47	81	26.08	17	156	3128.36	3.5	0

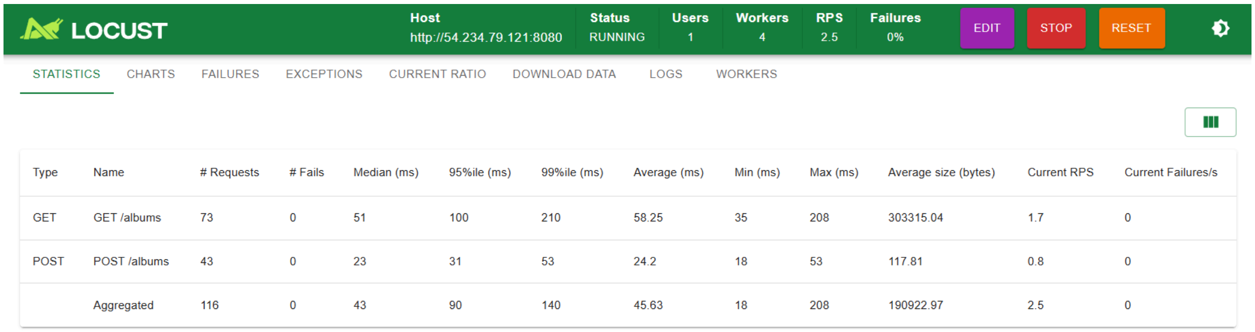
Are the throughputs making sense to you?

Yes—total is about 3.5 RPS (GET ~2.3, POST ~1.2), and even though my task ratio is 3:1, POST still takes a chunk of time so the RPS ratio isn’t exactly 3:1.

If we assign more workers to do work, how do you think the throughput will change? Linearly?

It should go up, but not perfectly linearly, because at some point the EC2 server (CPU/network/app code) may become the bottleneck and extra workers give smaller gains.

Let's find out. Increase worker count to 4 and record the statistics again.



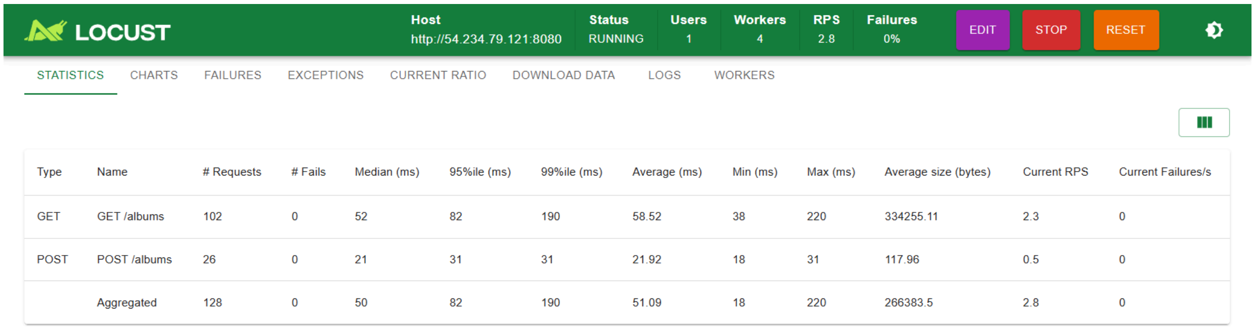
Host http://54.234.79.121:8080 | Status RUNNING | Users 1 | Workers 4 | RPS 2.5 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	73	0	51	100	210	58.25	35	208	303315.04	1.7	0
POST	POST /albums	43	0	23	31	53	24.2	18	53	117.81	0.8	0
	Aggregated	116	0	43	90	140	45.63	18	208	190922.97	2.5	0

Is there any way in which the reading and writing of the same item to something like a hashmap might contribute to your observations?

Yes, because a Go map/hashmap isn’t safe for concurrent reads/writes, lots of GETs and POSTs hitting it at the same time can cause contention (or even crashes with concurrent map writes), which can slow requests down and make latency worse when you add workers.

Context Switching



Host http://54.234.79.121:8080 | Status RUNNING | Users 1 | Workers 4 | RPS 2.8 | Failures 0%

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	102	0	52	82	190	58.52	38	220	334255.11	2.3	0
POST	POST /albums	26	0	21	31	31	21.92	18	31	117.96	0.5	0
	Aggregated	128	0	50	82	190	51.09	18	220	266383.5	2.8	0

What do you observe?

Switching from HttpUser to FastHttpUser with 4 workers only bumps total throughput a little (about 2.5 RPS → ~2.8 RPS), so it’s not a huge change. However, I believe the requests felt a bit faster.

Document your results, do some research, and try explaining the reasons behind it!

The GET /albums stays around ~58 ms avg but is slower than POST, while POST /albums stays faster at ~22–24 ms avg, and GET response size is massive because the album list keeps growing during the test. Why this happens could be that FastHttpUser reduces load-generator CPU overhead, but you’re mostly bottlenecked by the server/network and by GET returning a bigger and bigger payload, so Amdahl’s Law kicks in and you don’t get a big linear speedup just by using a faster client.