Part II - Thread Experiments

A) Atomicity (atomic int vs regular int)

Atomic integer

```go
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

func main() {
    start := time.Now()

    var ops atomic.Uint64

    var wg sync.WaitGroup

    for range 50 {
        wg.Go(func() {
            for range 1000 {
                ops.Add(1)
            }
        })
    }

    wg.Wait()

    fmt.Println("ops:", ops.Load())
    fmt.Println("time taken:", time.Since(start))
}
// --- OUTPUT ---
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 50000
// time taken: 3.10975ms
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 50000
// time taken: 3.60775ms
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 50000
// time taken: 3.590709ms
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 50000
// time taken: 3.644417ms
```

Regular integer

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    start := time.Now()

    var ops int64

    var wg sync.WaitGroup

    for range 50 {
        wg.Go(func() {
            for range 1000 {
                ops += 1
            }
        })
    }

    wg.Wait()

    fmt.Println("ops:", ops)
    fmt.Println("time taken:", time.Since(start))
}
// --- OUTPUT ---
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 31652
// time taken: 271.041µs
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 29285
// time taken: 96.459µs
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 30902
// time taken: 129.042µs
// mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run
atomic-counter.go
// ops: 29259
// time taken: 105.458µs
```

- **Observed results (4 runs)**:
    - Regular Counter final values are: 31652, 29285, 30902, 29259
    - Atomic Counter final values are: 50000, 50000, 50000, 50000
- **What's happening**: race condition in regular increment; atomic prevents lost updates.

- **-race flag result:**

```
mayanktamakuwala@Mayanks-MacBook-Pro Homework-3 % go run -race atomic-counter.go
==================
WARNING: DATA RACE
Read at 0x00c0000100e8 by goroutine 11:
  main.main.func1()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:21 +0x38
  sync.(*WaitGroup).Go.func1()
      /usr/local/go/src/sync/waitgroup.go:239 +0x54

Previous write at 0x00c0000100e8 by goroutine 8:
  main.main.func1()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:21 +0x48
  sync.(*WaitGroup).Go.func1()
      /usr/local/go/src/sync/waitgroup.go:239 +0x54

Goroutine 11 (running) created at:
  sync.(*WaitGroup).Go()
      /usr/local/go/src/sync/waitgroup.go:237 +0x78
  main.main()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:18 +0x84

Goroutine 8 (finished) created at:
  sync.(*WaitGroup).Go()
      /usr/local/go/src/sync/waitgroup.go:237 +0x78
  main.main()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:18 +0x84
==================
==================
WARNING: DATA RACE
Write at 0x00c0000100e8 by goroutine 14:
  main.main.func1()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:21 +0x48
  sync.(*WaitGroup).Go.func1()
      /usr/local/go/src/sync/waitgroup.go:239 +0x54

Previous write at 0x00c0000100e8 by goroutine 11:
  main.main.func1()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:21 +0x48
  sync.(*WaitGroup).Go.func1()
      /usr/local/go/src/sync/waitgroup.go:239 +0x54

Goroutine 14 (running) created at:
  sync.(*WaitGroup).Go()
      /usr/local/go/src/sync/waitgroup.go:237 +0x78
  main.main()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:18 +0x84

Goroutine 11 (running) created at:
  sync.(*WaitGroup).Go()
      /usr/local/go/src/sync/waitgroup.go:237 +0x78
  main.main()
      /Users/mayanktamakuwala/Documents/PHI/GitHub/Distributed-Systems/Homework-3/atomic-counter.go:18 +0x84
==================
ops: 29816
time taken: 2.561375ms
Found 2 data race(s)
exit status 66
```

- Running go run -race atomic-counter.go reported two data races at the same memory address, pointing to atomic-counter.go:21 inside the goroutine. The first warning shows a read occurring concurrently with a write (read/write race), and the second shows concurrent writes (write/write race). This happens because the shared counter is updated using ops++, which is not atomic (it performs a read-modify-write sequence). As a result, updates are lost when goroutines interleave, producing an incorrect final count (ops: 29816 instead of the expected 50000). The race detector then terminates with a non-zero exit code (exit status 66) to indicate unsafe concurrent access.

B) Collections - plain map[int]int with 50 goroutines

- Observed:
  - Program Crashed. Got following error

```
1. fatal error: concurrent map writes
```

- Reason:

  Go's normal map cannot be safely shared between goroutines. When multiple goroutines try to write to the same map at the same time, the program crashes with "fatal error: concurrent map writes". This happens because Go maps are not thread-safe. Without synchronization, the internal structure of the map becomes corrupted. This experiment shows that shared data must always be protected when accessed concurrently.

C) Mutex

- Result: len(m)=50000 consistently, time = 16.29 ms avg
- Lesson: Using a Mutex prevents crashes and guarantees correct results by allowing only one goroutine to access the map at a time. This makes the program safe and ensures that all 50,000 entries are inserted correctly. However, because every goroutine must wait its turn, performance decreases when many goroutines compete for the same lock. Mutex provides correctness but can become a bottleneck under heavy concurrency.

D) RWMutex

- Result: len(m)=50000 consistently, time = 15.81 ms avg
- Lesson: RWMutex allows multiple goroutines to read data at the same time, but still allows only one writer. In this experiment, almost all operations were writes, so RWMutex behaved very similarly to a regular Mutex. Since there were few or no reads, the extra complexity of RWMutex did not provide any performance improvement. RWMutex is mainly useful when programs have many readers and very few writers.

E) Sync.Map

- Result: len(m)=50000 consistently, time = 7.027 ms avg
- Lesson: sync.Map is designed specifically for concurrent access and does not require manual locking. It handled all inserts correctly and avoided crashes. In some cases, sync.Map can perform better than Mutex because it reduces lock contention internally. However, it has higher overhead and is best suited for specific patterns such as read-heavy workloads. It is not always faster than Mutex and should be used carefully.

F) File Access

- Unbuffered: 106.417885ms avg
- Buffered: 309.260875µs avg

- Lesson: Writing to a file without buffering causes each write to be handled individually by the operating system, resulting in many system calls and slower performance. Buffered writing groups many small writes into larger chunks in memory before sending them to disk, significantly reducing system overhead. As a result, buffered file writing is much faster. This experiment demonstrates how minimizing system calls through buffering can greatly improve I/O performance.

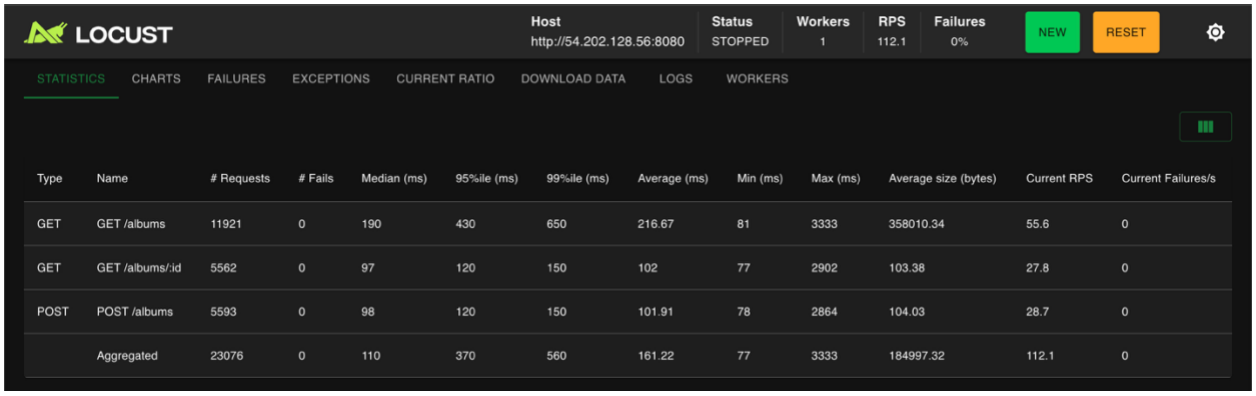
G) Context Switching

- One Thread: 22.9298545ms avg
- Multiple Threads: 23.2085µs avg
- Lesson: This experiment measures the cost of handing control between two goroutines by passing a token back and forth on an unbuffered channel many times. With GOMAXPROCS(1), only one OS thread runs Go code at a time, so the goroutines must take turns, which increases total time. With multiple OS threads enabled, the scheduler has more flexibility and the handoff loop can run more efficiently, reducing time. Even so, this is still much lighter than switching between processes/containers/VMs, because goroutine switching happens mostly inside the Go runtime, while process/VM switching involves heavier OS and hardware overhead.

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

1) Locust Statistics Page

- 1 worker

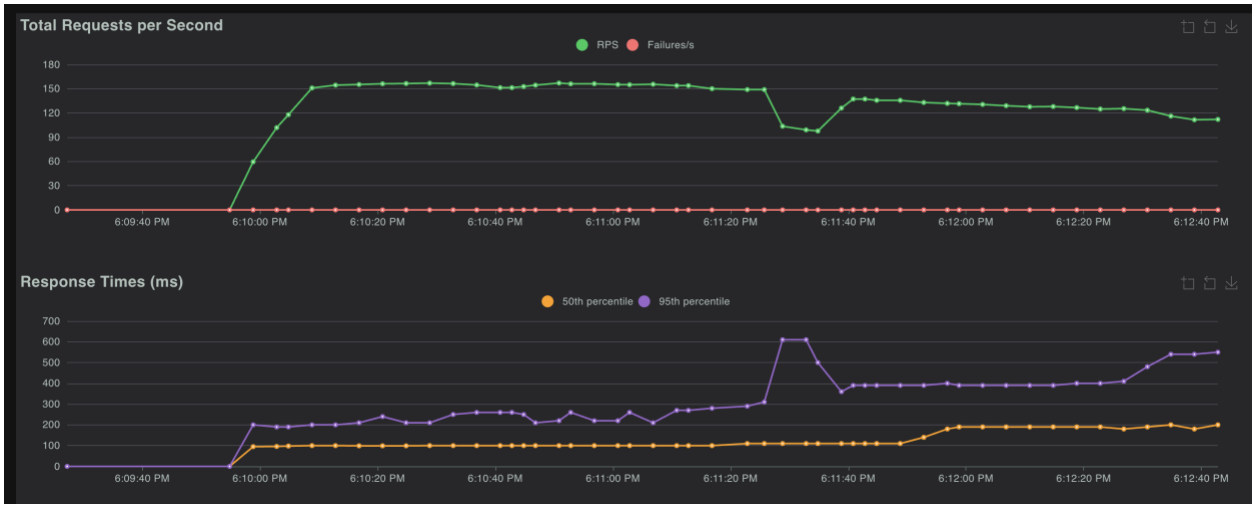


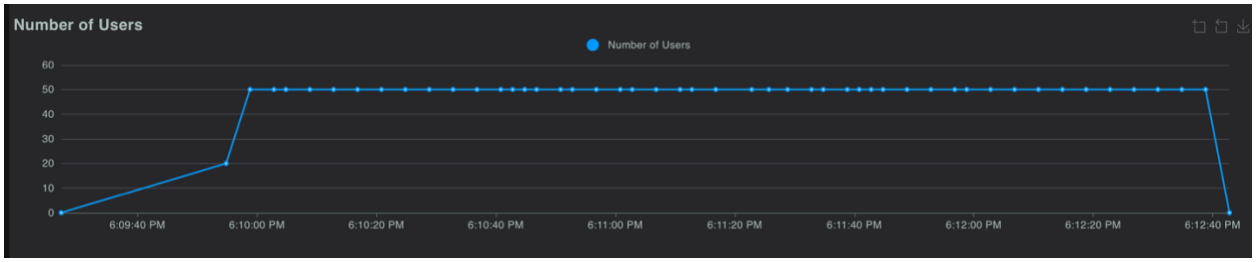
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	11921	0	190	430	650	216.67	81	3333	358010.34	55.6	0
GET	GET /albums/:id	5562	0	97	120	150	102	77	2902	103.38	27.8	0
POST	POST /albums	5593	0	98	120	150	101.91	78	2864	104.03	28.7	0
	Aggregated	23076	0	110	370	560	161.22	77	3333	184997.32	112.1	0

- 3 workers

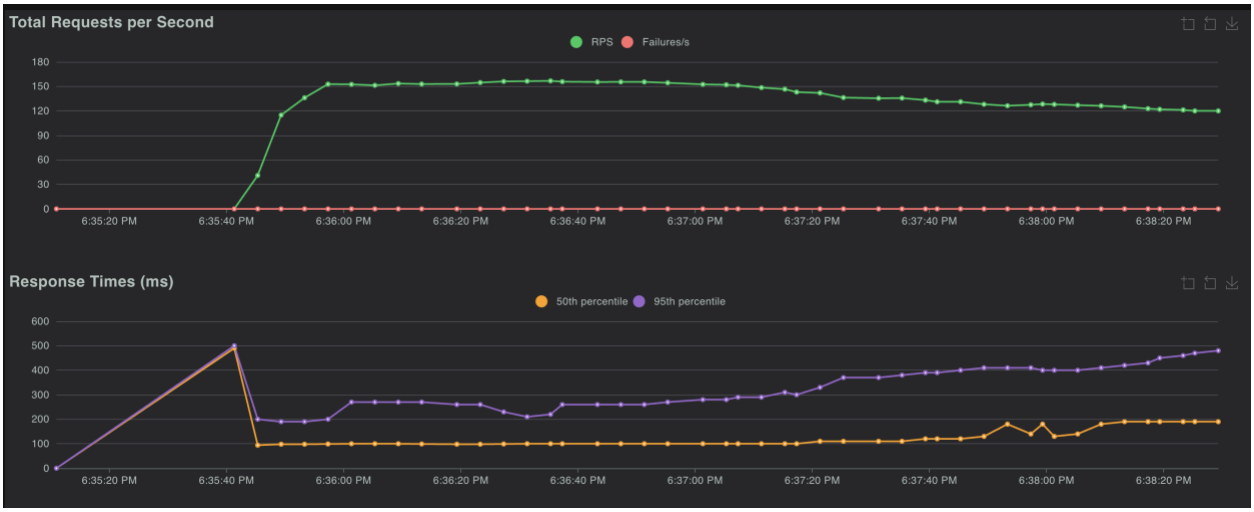
2) Locust Charts Page

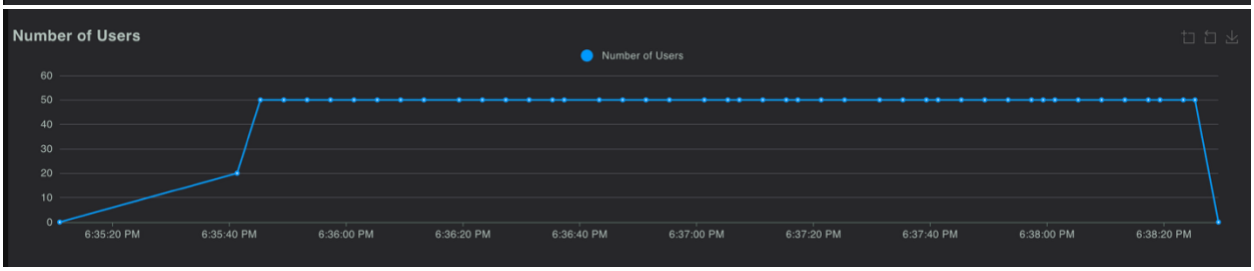
- 1 worker





- 3 workers





3) Docker Stats

- 1 worker

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
25d0c34f9979	homework-3-locust-worker-1	0.28%	238MiB / 7.653GiB	3.04%	4.42GB / 36.9MB	0B / 0B	3
2315b52197c3	homework-3-locust-master-1	0.16%	35.77MiB / 7.653GiB	0.46%	420kB / 1.6MB	0B / 0B	3
f0dac23956b7	buildx_buildkit_multi0	0.08%	131.5MiB / 7.653GiB	1.68%	53.9kB / 44kB	91.9MB / 5.35MB	35

- 3 workers:

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
c861c2bcaab9	homework-3-locust-master-1	0.28%	35.94MiB / 7.653GiB	0.46%	678kB / 496kB	0B / 0B	3
8213c39255db	homework-3-locust-worker-1-1	0.15%	114MiB / 7.653GiB	1.45%	1.52GB / 15.8MB	0B / 0B	3
dc9e747140ed	homework-3-locust-worker-2-1	0.19%	96.02MiB / 7.653GiB	1.23%	1.45GB / 11.9MB	0B / 0B	3
42dc482602ea	homework-3-locust-worker-3-1	0.19%	111.5MiB / 7.653GiB	1.42%	1.53GB / 12.4MB	0B / 0B	3
f0dac23956b7	buildx_buildkit_multi0	0.02%	129.8MiB / 7.653GiB	1.66%	90.7kB / 69.6kB	92MB / 7.35MB	35

Locust was used to perform load testing on the album API using one worker, 50 concurrent users, and a ramp-up rate of 10 users per second with a 3:1 GET to POST request ratio. During the experiment, a total of 23,076 requests were processed with zero failures, indicating stable server behavior under load.

For GET /albums, the average response time was approximately 216 ms with a 95th percentile of 430 ms. This endpoint exhibited the highest latency because it returns the full album collection, resulting in larger payload sizes and increased serialization overhead.

For GET /albums/:id, the average response time was approximately 102 ms with a 95th percentile of 120 ms. This endpoint is significantly faster since it retrieves only a single album record.

For POST /albums, the average response time was approximately 102 ms with a 95th percentile of 120 ms. Although POST requests modify server state, their response size is small and comparable to GET /albums/:id, leading to similar latency.

The system achieved an aggregated throughput of approximately 112 requests per second. Percentile metrics were used in addition to averages to better capture tail latency, as averages alone can hide slower requests that negatively impact user experience.

System resources were monitored using docker stats during testing to ensure that CPU and memory usage remained within reasonable limits

Amdahl's Law

- With one worker, the system achieved approximately 112 requests per second with an average latency of 161 ms. After increasing the worker count to three, throughput increased slightly to approximately 120 requests per second, while average latency decreased modestly to 154 ms. This represents only a small performance improvement rather than the expected near-linear scaling.

- This behavior is explained by Amdahl's Law, which states that overall system speedup is limited by the portion of the workload that must execute serially. In this application, concurrent reads and writes to the shared in-memory album slice introduce contention and synchronization overhead. POST requests modify shared state, while GET requests access the same data structure, creating serialization points that restrict parallel execution.

- As a result, adding more workers provided diminishing returns instead of proportional throughput gains. This experiment demonstrates that shared data structures, CPU limits, and coordination overhead significantly constrain scalability, highlighting the importance of minimizing shared state and optimizing critical sections in concurrent and distributed systems.

Context Switching (FastHttpUser)

- By default, Locust uses the Python requests library, which introduces overhead when generating high request rates. To reduce client-side bottlenecks, HttpUser was replaced with FastHttpUser, which uses a C-based HTTP client optimized for high concurrency.
- After switching to FastHttpUser (with three workers), the system initially achieved a peak throughput of approximately 156 requests per second, compared to approximately

120 requests per second using HttpUser. This indicates that the original load generator was partially limiting performance. However, as the total request count increased to around 23,000, throughput gradually declined and stabilized near 118 requests per second, while average latency remained roughly unchanged.

- This behavior shows that once client-side overhead was removed, the server became the primary bottleneck. Shared access to the in-memory album slice, increased payload sizes for GET /albums as data grew, and CPU contention introduced serialization points that limited sustained throughput. Although FastHttpUser reduced context-switching and networking overhead on the client side, overall performance was ultimately constrained by server-side resources and synchronization costs.
- This experiment demonstrates that improving the efficiency of the load generator can temporarily increase throughput, but long-term scalability is governed by server-side limits, reinforcing Amdahl's Law and the impact of shared state in concurrent systems.