

# CS 6650 Assignment 3: Thread Experiments & Load Testing Report

Eroniction Presley

## 1 Part II: Thread Experiments (Go)

### 1.1 Atomicity

**Concept:** Atomic operations guarantee that read-modify-write operations complete without interference from other threads.

**Experiment:** Compared atomic counter vs regular counter with 50 goroutines, each incrementing 1000 times.

**Results:**

Run	Atomic Counter	Regular Counter
1	50000	50000
2	50000	50000
3	50000	50000
4	50000	48302

Table 1: Atomicity Experiment Results

```
PS D:\BSDS\Assignment 3> go run atomicity.go
Atomic counter: 50000
Regular counter: 50000
PS D:\BSDS\Assignment 3> go run atomicity.go
Atomic counter: 50000
Regular counter: 50000
PS D:\BSDS\Assignment 3> go run atomicity.go
Atomic counter: 50000
Regular counter: 50000
PS D:\BSDS\Assignment 3> go run atomicity.go
Atomic counter: 50000
Regular counter: 48302
```

Figure 1: Atomicity Experiment Terminal Output

**Analysis:** The atomic counter always returns the correct value (50,000). The regular counter lost 1,698 increments (3.4%) due to a race condition where multiple goroutines read the same value before incrementing, causing lost updates. This demonstrates why atomic operations are essential for shared state in concurrent programs.

### 1.2 Collections (Plain Map)

**Concept:** Go's built-in maps are not thread-safe and will crash under concurrent writes.

**Experiment:** 50 goroutines writing 1000 entries each to a shared map.

**Results:**

```
fatal error: concurrent map writes
```

```
PS D:\BSDS\Assignment 3> go run collections.go  
fatal error: concurrent map writes
```

Figure 2: Collections Crash Output

**Analysis:** Unlike the atomicity experiment which silently lost data, Go's map deliberately panics when it detects concurrent writes. This is a safety feature—Go crashes rather than silently corrupting data. Maps have internal structure (buckets, hash tables) that can get corrupted if modified simultaneously.

### 1.3 Mutex

**Concept:** A mutex (mutual exclusion lock) ensures only one goroutine can access a resource at a time.

**Experiment:** Wrapped the map in a struct with `sync.Mutex`, locking before each write.

**Results:**

Run	Map Length	Time
1	50000	2.29ms
2	50000	6.85ms
3	50000	5.66ms
4	50000	5.75ms
5	50000	6.11ms

Table 2: Mutex Experiment Results (Average:  $\sim 5.3$ ms)

```
● PS D:\BSDS\Assignment 3> go run mutex.go  
Map length: 50000  
Time taken: 2.2944ms  
● PS D:\BSDS\Assignment 3> go run mutex.go  
Map length: 50000  
Time taken: 6.8473ms  
● PS D:\BSDS\Assignment 3> go run mutex.go  
Map length: 50000  
Time taken: 5.6635ms  
● PS D:\BSDS\Assignment 3> go run mutex.go  
Map length: 50000  
Time taken: 5.7502ms  
● PS D:\BSDS\Assignment 3> go run mutex.go  
Map length: 50000  
Time taken: 6.1144ms
```

Figure 3: Mutex Experiment Terminal Output

**Analysis:** The mutex guarantees correctness—all 50,000 entries are written successfully. However, goroutines must wait in line to acquire the lock, serializing access to the map.

## 1.4 RWMutex

**Concept:** RWMutex allows multiple concurrent readers OR one exclusive writer.

**Experiment:** Replaced sync.Mutex with sync.RWMutex using Lock() for writes.

**Results:**

Run	Map Length	Time
1	50000	7.80ms
2	50000	9.58ms
3	50000	7.56ms
4	50000	6.00ms

Table 3: RWMutex Experiment Results (Average: ~7.7ms)

```
PS D:\BSDS\Assignment 3> go run rwmutex.go
Map length: 50000
Time taken: 7.8015ms
PS D:\BSDS\Assignment 3> go run rwmutex.go
Map length: 50000
Time taken: 9.5754ms
PS D:\BSDS\Assignment 3> go run rwmutex.go
Map length: 50000
Time taken: 7.5585ms
PS D:\BSDS\Assignment 3> go run rwmutex.go
Map length: 50000
Time taken: 6.0038ms
```

Figure 4: RWMutex Experiment Terminal Output

**Analysis:** RWMutex is actually **slower** than regular Mutex for write-heavy workloads. The extra bookkeeping for tracking read vs write locks adds overhead. RWMutex only benefits workloads with many readers and few writers, where readers can proceed concurrently with RLock().

## 1.5 sync.Map

**Concept:** sync.Map is Go's built-in concurrent-safe map, optimized for specific access patterns.

**Experiment:** Used sync.Map with Store() method for writes.

**Results:**

**Analysis:** sync.Map performs comparably to Mutex. It's optimized for cases where keys are written once and read many times, or where goroutines access disjoint key sets. Not always faster than Mutex + map.

## 1.6 Collections Summary

## 1.7 File Access

**Concept:** Buffered I/O batches writes to reduce expensive system calls.

Run	Map Length	Time
1	50000	8.80ms
2	50000	5.04ms
3	50000	3.63ms
4	50000	4.98ms

Table 4: sync.Map Experiment Results (Average:  $\sim 5.6$ ms)

```

● PS D:\BSDS\Assignment 3> go run syncmap.go
  Map length: 50000
  Time taken: 8.7958ms
● PS D:\BSDS\Assignment 3> go run syncmap.go
  Map length: 50000
  Time taken: 5.0388ms
● PS D:\BSDS\Assignment 3> go run syncmap.go
  Map length: 50000
  Time taken: 3.6334ms
● PS D:\BSDS\Assignment 3> go run syncmap.go
  Map length: 50000
  Time taken: 4.9775ms
○ PS D:\BSDS\Assignment 3> █

```

Figure 5: sync.Map Experiment Terminal Output

**Experiment:** Compared unbuffered vs buffered writes for 100,000 iterations.

**Results:**

```

PS D:\BSDS\Assignment 3> go run fileaccess.go
Unbuffered time: 165.9997ms
Buffered time: 6.0053ms
Buffered is 27.6x faster
PS D:\BSDS\Assignment 3> go run fileaccess.go
Unbuffered time: 167.489ms
Buffered time: 3.0757ms
Buffered is 54.5x faster
PS D:\BSDS\Assignment 3> go run fileaccess.go
Unbuffered time: 186.9699ms
Buffered time: 3.6165ms
Buffered is 51.7x faster

```

Figure 6: File Access Experiment Terminal Output

**Analysis:** Buffered writes are **27–55x faster**. Unbuffered mode makes 100,000 system calls (one per write), while buffered mode collects writes in memory and flushes in large chunks. This principle applies to distributed systems—network calls are expensive, so batching requests improves performance.

## 1.8 Context Switching

**Concept:** Switching between goroutines has a cost that varies based on thread configuration.

**Experiment:** Measured ping-pong communication between two goroutines over 1 million iterations.

Approach	Avg Time	Thread-Safe	Best Use Case
Plain map	CRASH	No	Single-threaded only
Mutex	~5.3ms	Yes	Write-heavy workloads
RWMutex	~7.7ms	Yes	Read-heavy workloads
sync.Map	~5.6ms	Yes	Write-once, read-many

Table 5: Comparison of Synchronization Approaches

Run	Unbuffered	Buffered	Speedup
1	166.00ms	6.01ms	27.6x
2	167.49ms	3.08ms	54.5x
3	186.97ms	3.62ms	51.7x

Table 6: File Access Experiment Results

## Results:

Configuration	Total Time	ns/switch
GOMAXPROCS=1	~295ms	~148 ns
GOMAXPROCS=22	~311ms	~156 ns

Table 7: Context Switching Experiment Results

**Analysis:** Single-thread switching is ~5% faster. With GOMAXPROCS=1, both goroutines run on the same OS thread and Go’s scheduler swaps them in memory (cheap). With multiple threads, channel communication may cross OS threads, requiring OS-level synchronization.

### Distributed Systems Implication:

This is why minimizing network round-trips matters in distributed systems.

## 2 Part III: Load Testing with Locust

### 2.1 Test Setup

- **Server:** Go/Gin REST API on port 8080
- **Endpoints:** GET /albums, POST /albums
- **Task ratio:** 3:1 (GET:POST)
- **Tool:** Locust with Docker

### 2.2 Warmup Test (1 user, 1 worker)

**Analysis:** POST is slower (32ms vs 11ms) because it parses JSON and modifies the slice. The 3:1 task ratio is reflected in request counts.

### 2.3 Load Test (50 users, 1 worker)

**Analysis:** Under load, GET became slower than POST (110ms vs 54ms) because the albums list keeps growing from POST requests, increasing response size to 26KB.

```

• PS D:\BSDS\Assignment 3> go run contextswitching.go
  GOMAXPROCS=1: 283.2615ms (142 ns/switch)
  GOMAXPROCS=22: 312.3052ms (156 ns/switch)
• PS D:\BSDS\Assignment 3> go run contextswitching.go
  GOMAXPROCS=1: 297.993ms (149 ns/switch)
  GOMAXPROCS=22: 311.506ms (156 ns/switch)
• PS D:\BSDS\Assignment 3> go run contextswitching.go
  GOMAXPROCS=1: 303.9697ms (152 ns/switch)
  GOMAXPROCS=22: 310.8031ms (155 ns/switch)

```

Figure 7: Context Switching Experiment Terminal Output

Switch Type	Approximate Cost
Goroutine (same thread)	~150 ns
OS thread context switch	~1,000–2,000 ns
Process context switch	~3,000–5,000 ns
Network call (nodes)	~500,000+ ns

Table 8: Context Switch Cost Comparison

## 2.4 Amdahl’s Law Test (50 users, 4 workers)

### Comparison: 1 Worker vs 4 Workers

**Amdahl’s Law Insight:** 4 workers  $\neq$  4x throughput. We got ~1.5x because the bottleneck shifted to the server (single Go process). Adding more load generators doesn’t help when the server is saturated.

## 2.5 FastHttpUser Test (50 users, 4 workers)

### Comparison: HttpUser vs FastHttpUser (4 Workers)

**Analysis:** FastHttpUser performed worse because:

1. It’s optimized for high concurrency (1000+ users), not 50 users
2. The overhead of the C-based library isn’t worth it at this scale
3. The bottleneck is the server, not Locust’s HTTP client

**Key Lesson:** Faster tools don’t always mean better performance. Choose tools based on your actual scale and bottleneck.

## 3 Key Takeaways

1. **Atomicity:** Use atomic operations or locks for shared state—regular operations cause silent data loss.
2. **Collections:** Go maps crash under concurrent writes. Choose synchronization based on access pattern (Mutex for writes, RWMutex for read-heavy, sync.Map for write-once).
3. **Buffering:** Batch I/O operations—27–55x improvement from buffered writes.
4. **Context Switching:** Goroutine switches are cheap (~150ns), but network calls are expensive (~500,000ns). Design distributed systems to minimize round-trips.

Type	Requests	Avg (ms)	Size (bytes)
GET	30	10.86	846
POST	7	32.58	93

Table 9: Warmup Test Results

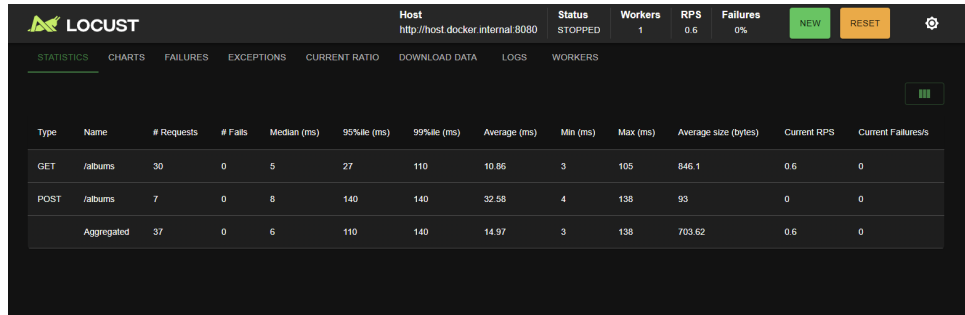


Figure 8: Locust Warmup Test Statistics

5. **Amdahl's Law:** Scaling isn't linear. Identify and address the actual bottleneck before adding resources.
6. **Tool Selection:** Match tools to your scale. FastHttpUser helps at 1000+ users, not 50.

Type	Requests	Avg (ms)	99%ile (ms)	RPS
GET	1162	110.3	3500	15.1
POST	442	53.7	1200	6.8
<b>Total</b>	<b>1604</b>	<b>94.7</b>	<b>3300</b>	<b>21.9</b>

Table 10: Load Test Results (1 Worker)

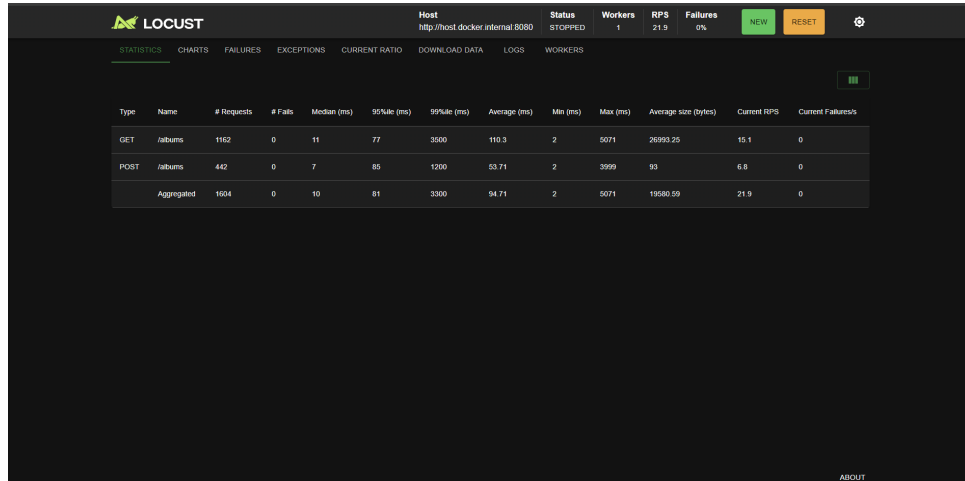


Figure 9: Locust 50 Users / 1 Worker - Statistics

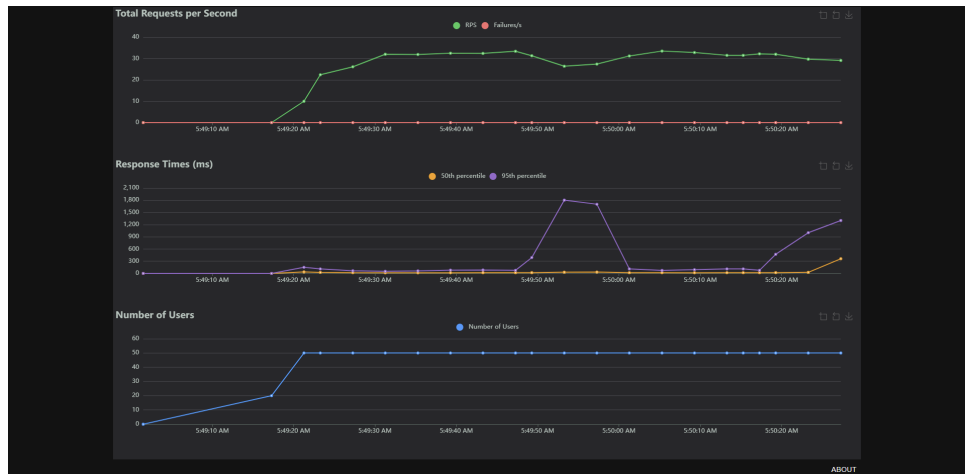


Figure 10: Locust 50 Users / 1 Worker - Charts

Type	Requests	Avg (ms)	99%ile (ms)	RPS
GET	1700	25.06	130	24.9
POST	591	8.61	50	7.5
<b>Total</b>	<b>2291</b>	<b>20.81</b>	<b>110</b>	<b>32.4</b>

Table 11: Load Test Results (4 Workers)

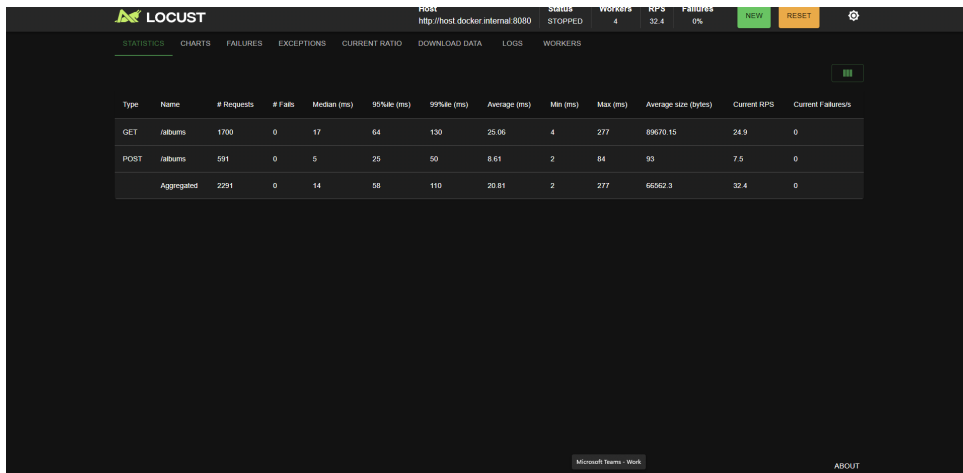


Figure 11: Locust 50 Users / 4 Workers - Statistics

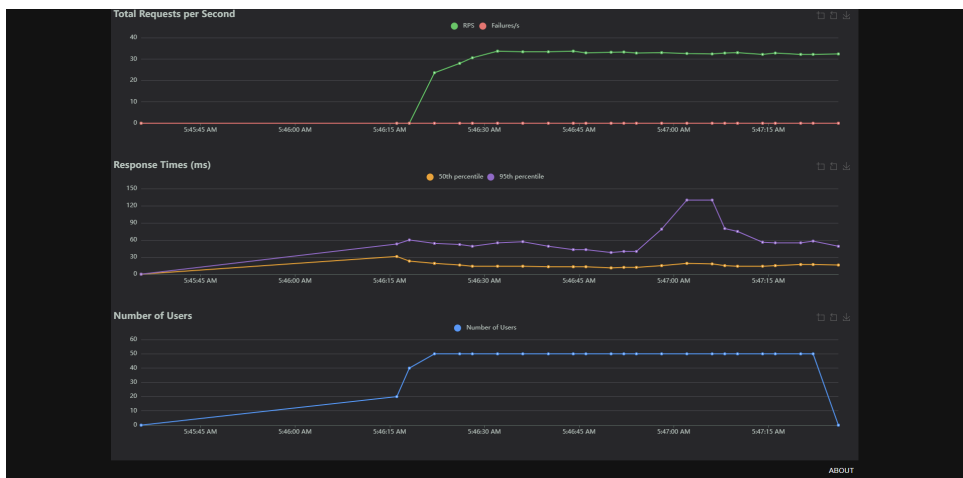


Figure 12: Locust 50 Users / 4 Workers - Charts

Metric	1 Worker	4 Workers	Improvement
Total RPS	21.9	32.4	+48%
GET Avg (ms)	110.3	25.06	4.4x faster
99%ile (ms)	3300	110	30x better

Table 12: Amdahl's Law Comparison

Type	Requests	Avg (ms)	99%ile (ms)	RPS
GET	1795	302.49	7700	18.7
POST	590	78.59	1500	9.4
<b>Total</b>	<b>2385</b>	<b>247.1</b>	<b>7200</b>	<b>28.1</b>

Table 13: FastHttpUser Test Results

Metric	HttpUser	FastHttpUser	Change
Total RPS	32.4	28.1	-13%
GET Avg (ms)	25.06	302.49	12x slower
99%ile (ms)	110	7200	65x worse

Table 14: HttpUser vs FastHttpUser Comparison

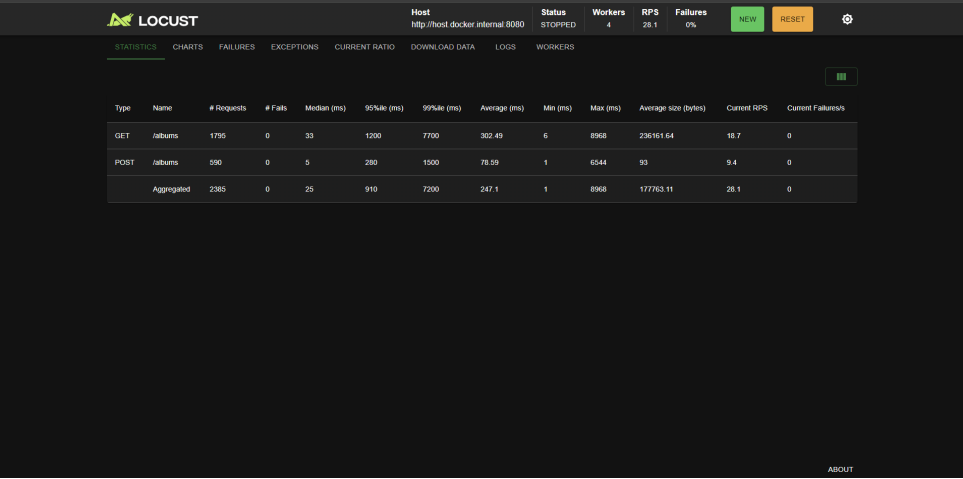


Figure 13: FastHttpUser - Statistics

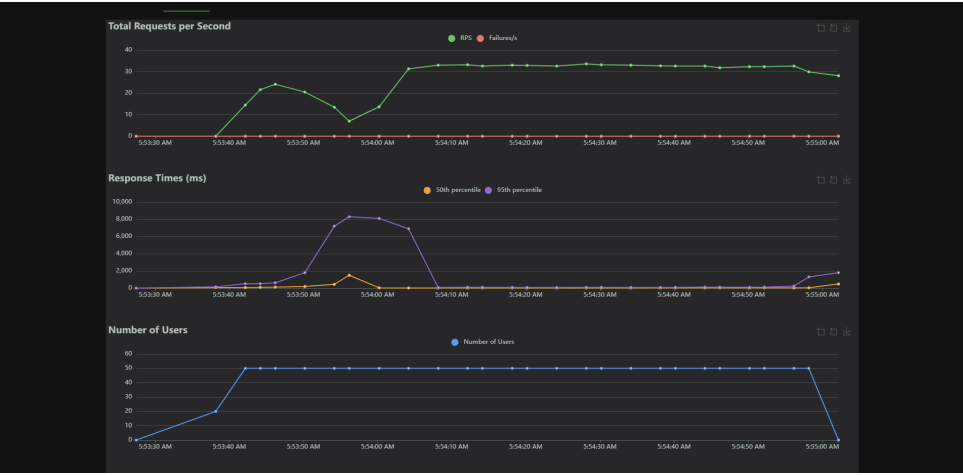


Figure 14: FastHttpUser - Charts