# Coordinated Omission Problem in UNH Cloud Serving Benchmark

Marielle Webster, Computer Science, University of New Hampshire

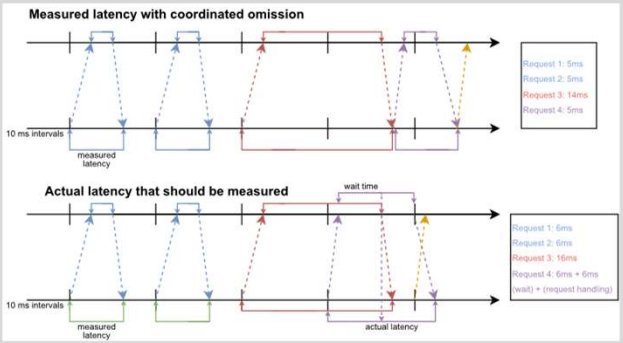
## BACKGROUND & MOTIVATION

### YCSB & MEASURING LATENCY

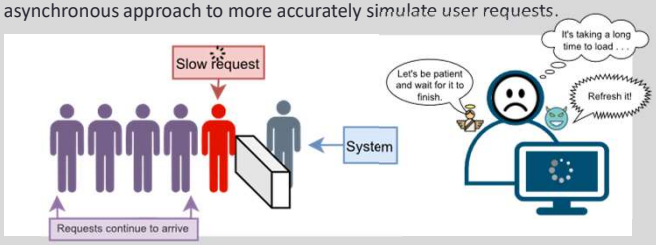
- The *Yahoo! Cloud Serving Benchmark* is an open-source project, originally released in 2010 to provide an extensible benchmark framework. The UNH Cloud Serving Benchmark is a fork of a Go implementation of this service.
- A workload generator is used to schedule requests to a system.
- Latency measurements are an important metric when testing cloud serving systems. Developers need to be able to *effectively* and *reliably* test systems to decide what framework is best suited to their needs.
- These measurements need to accurately reflect the user experience.

### COORDINATED OMISSION PROBLEM

The Coordinated Omission Problem is a common flaw in latency measurement tools where the measuring system accidentally coordinates with the system under test to omit the effect of outliers – requests that take longer for a system to handle.



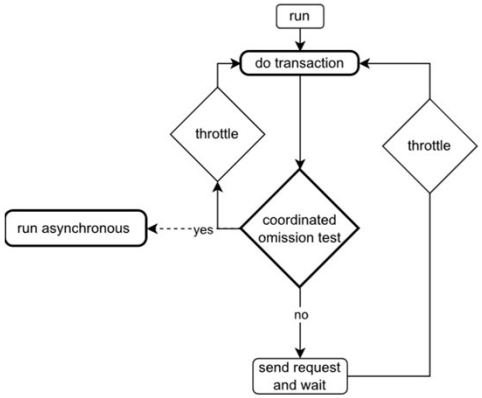

In the first example, the workload generator waits for the request to complete before sending the next one. This type of synchronization is useful in closed systems where a new job can't arrive until the current job completes. However, in open systems where requests can arrive at any given time, such as with cloud-based applications, we need an asynchronous approach to more accurately simulate user requests.




## IMPLEMENTATION

In our Go-extended implementation of YCSB, we navigate this fix by using goroutines – lightweight threads managed by the Go runtime that runs concurrently. An additional parameter was added to the workload generator in order to make this a toggleable solution.

- Each goroutine handles its own request asynchronously from the worker thread which represents a client making requests to the system.




```
run(): worker thread
WHILE running
      do transaction
            if coordinatedOmissionFix
                  asyncTransaction(): new thread
            else
                  transaction()
      add to requestsSent
      throttle()
```

```
throttle(): worker thread
requestStartTime = requestNum*nanosecondsPerRequest
waitUntil(requestStartTime)
```

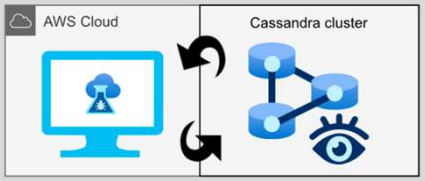
Impacts:

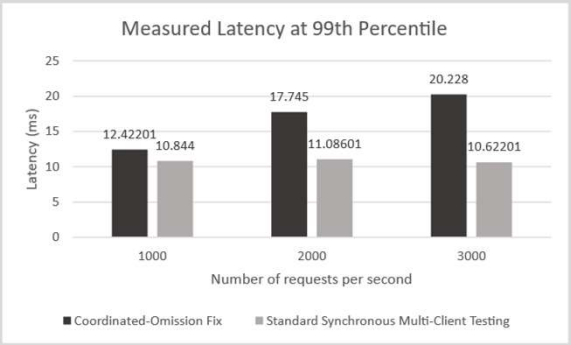
- Slow requests with high throughput may lead to excessive thread creation as requests are generated without completion and slow benchmark system down.
- Worker threads are not safe for concurrent access against self. Cannot send two requests simultaneously. May cause issues if a large target is specified without a reasonable number of clients to distribute the work between.

## TESTING & RESULTS

Testing methods:

- ❑ Same workload ran with original synchronous implementation and asynchronous fix.
- ❑ 3 different targets of requests per second. 20 worker threads.
- ❑ 3-node cluster of Cassandra database hosted on AWS.




## CONCLUSIONS

- ✓ Tail latency increases with throughput.
- ✓ More accurate measurement of tail latency.
- ✓ Better idea of how delays in processing requests effect system performance and impact users.
- ✓ More informed decisions regarding distributed system usage and expectations.