Can Java microservices be as fast as Go?

Mark Nelson
Nov 5 · 11 min read

Written by Mark Nelson and Peter Nagy.

Peter Nagy and I presented a paper at the Oracle Groundbreakers Tour 2020 LATAM in August, 2020 titled “Go Java, Go!” where we asked ourselves the question “can Java microservices be as fast as Go?” We created some microservices and did some benchmarking, and we presented our results at that event. But there was more to explore, so we decided to turn our presentation into this post. We plan to follow up with additional posts as we do more testing and development.

Premise

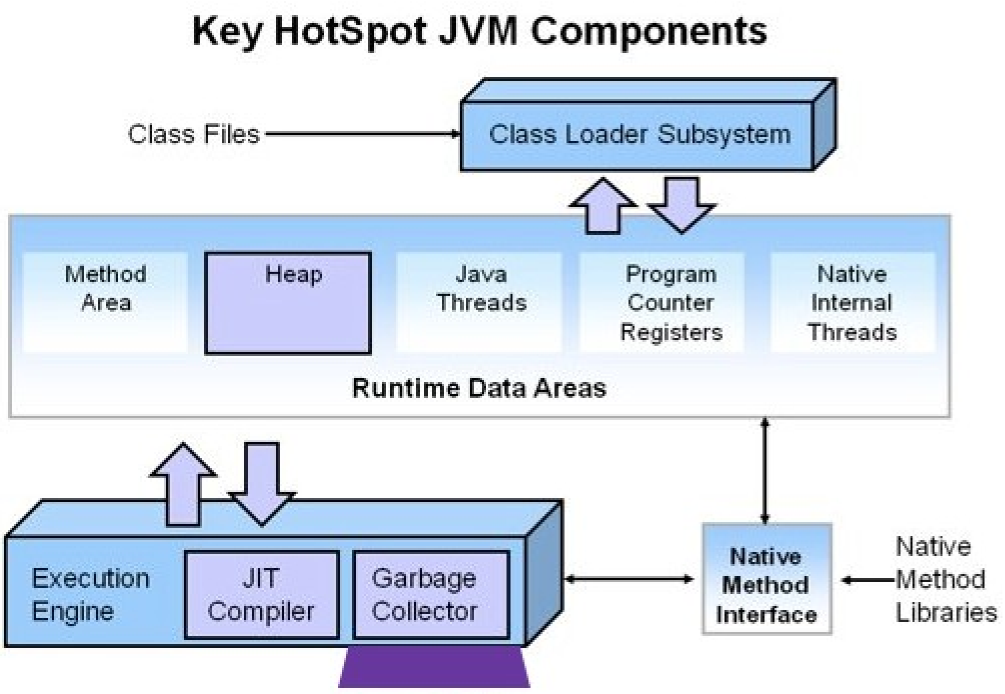
We wanted to experiment to see if Java microservices could run as fast as Go microservices. Common industry belief seems to be that Java is “old,” “slow” and “boring;” and that Go is “fast,” “new” and “cool.” But we wanted to know if those characterizations were warranted or supported by actual performance data.

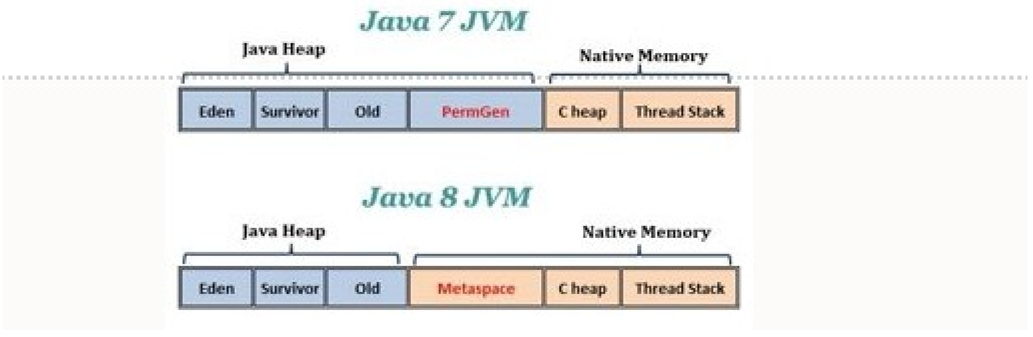
We wanted a fair test, so we created a very simple microservice with no external dependencies (like a database for example), and with very short code paths (just manipulating strings). We did include metrics and logging, since these seem to be always included in any real microservice. We used small, lightweight frameworks (Helidon for Java and Go-Kit for Go) and we also experimented with pure JAX-RS for Java. We experimented with different versions of Java and different JVMs. We did some basic tuning of the heap size and garbage collector. We warmed up the microservices before test runs.

A little history — Java

Java was developed by Sun Microsystems, who were acquired by Oracle. Its 1.0 release was in 1996, and the latest release was Java 15 in 2020. The main design goals are portability of the Java virtual machine and bytecode, and memory management with garbage collection. It is still one of the most popular languages (according to sources like StackOverflow and TIOBE), it is developed in open source.

Let’s talk about the “problem with Java.” It has a reputation for being slow, which is probably not justified any more but more historical. It does have some performance sensitive areas including the heap, where object data are stored; the garbage collector, which manages the heap; and the just in time (JIT) compiler.





Java has had a number of different garbage collection algorithms over the years, including serial, parallel, concurrent mark/sweep, G1 and the new ZGC garbage collector. Modern garbage collectors aim to minimize the duration of garbage collection “stop the world” pauses.

Oracle Labs have developed a new Java Virtual Machine called GraalVM which is written in Java and has a new compiler and some exciting new features like the ability to convert Java byte code into a native image that can be run without a Java VM.

A little history — Go

Go was created by Robert Griesemer, Rob Pike and Ken Thomson at Google. Between them they made major contributions to UNIX, B, C, Plan9, UNIX windowing system, and more.) It is open source, had its 1.0 release in 2012 and had its 1.15 release in 2020. It is growing fast both in terms of adoption and the language and tooling ecosystem itself.

Go is influenced by C, Python, Javascript and C++. It is designed to be a best of breed language for high performance networking and multiprocessing.

StackOverflow had 27,872 questions tagged with “Go” when we presented this talk, compared to 1,702,730 for Java.

Go is a statically-typed, compiled language. Its syntax is C-like. It has memory safety, garbage collection, structural typing, and CSP-style concurrency (communicating sequential processes). It has lightweight processes called goroutines (these are not OS threads), channels for communicating between them (typed, FIFO). The language does not provide race condition protection.

Go is the language of choice for many CNCF projects, for example Kubernetes, Istio, Prometheus and Grafana are all written (largely) in Go.

It is designed to have fast build time and fast execution. It is opinionated — no more arguing about two or four spaces!

What’s good about Go (compared to Java) — this is my personal opinion based on my experience:

- It is easier to implement functional patters like composition, pure functions, immutable state.
- There is much less boilerplate code (but still too much).
- It is still early in its lifecycle so it does not have the same crushing burden of backward compatibility — they can still break things to improve it.
- It compiles into a native statically-linked binary — no virtual machine layer — the binary has everything you need to run the program, which is great for “FROM scratch” containers.
- It has small size, fast startup and fast execution.
- No OOP, inheritance, generics, assertions, pointer arithmetic,.
- Less parentheses, e.g. `if x > 3 { whatever }`
- Enforces that are no cyclic dependencies, no unused variables or imports, no implicit type conversions.

So, what are the “problems” with Go? Again, this is my personal opinion, compared to Java:

- The tools ecosystem is immature, especially dependency management — there were several options, none of them were perfect, especially for non-open source development; now there is a clear “winner” (Go modules) but not everyone has adopted it, so there are still compatibility challenges.
- Building code with new/updated dependencies is very slow (like Maven’s famous “download the Internet” problem.
- Imports tie the code to the repository which makes moving code around a nightmare.
- IDEs are good for programming, documentation lookup, auto-complete, and so on; but debugging, profiling, and the like are still challenging.
- Pointers! We thought we left them back in the last millennium! But at least there is not pointer arithmetic.
- There is no Java-style try/catch for exceptions (you end up writing `if err != nil` way too often), no functional-style primitives like lists, map function, etc.

- You often end up implementing some basic algorithm since it is just no available yet. Recently I wrote code that walked through two strings (lists) slot by sloe doing comparisons and transformations. In a functional language I could have used built-ins like `map` to do that.
- There no is dynamic linking! ("Who cares?" you ask) It can be a real problem if you want to use code with licenses like GPL that "infect" statically linked code.
- There are not many knobs to tune execution or garbage collection, profile execution or optimization algorithms — Java has hundreds of garbage collection tuning options, Go has one — on or off.

Load testing methodology

We used JMeter to run our load tests. The tests call the services many times and collect data about response time, throughput (transactions per second) and memory usage. For Go we collect the resident set size and for Java we tracked native memory.

In many of the tests we ran JMeter on the same machine as the application under test. There did not seem to be any interference or difference in results if we ran JMeter on a different machine, so this simplified the setup. When we later deployed the applications into Kubernetes, JMeter was running on a remote machine, outside the cluster.

Before measuring, we warmed up the application using 1,000 invocations of the services.

The source code for the applications themselves, and the definitions of the load tests are all in this GitHub repository: https://github.com/markxnelson/go-java-go

First round of testing

In the first round, we ran the test on a "small" machine, in this case a 2.5GHz dual-core Intel Core i7 laptop with 16GB of RAM running macOS. We ran 100 threads with 10,000 loops per thread and a 10 second ramp up time. Java applications ran on JDK 11 and Helidon 2.0.1. Go applications compiled with Go 1.13.3.

The results were as follows:

We declared Go the winner of the first round!

Here are our observations from these results:

- Logging seems to be a major performance hit, especially java.util.logging. Because of this, we ran tests with and without logging. We also noticed that logging was a significant factor in the performance of the Go applications.
- The Java versions had a significantly larger memory footprint, even for such a small and simple application
- Warmup made a big difference for the JVM — we know that the JVM does optimizations as it runs so this makes sense
- We are comparing different execution models in this test — the Go application was compiled into a natively executable binary whereas the Java application was compiled into byte code that was then run on a virtual machine. We decided to introduce GraalVM native image so bring the Java application's execution environment much closer to the Go application's environment.

GraalVM native image

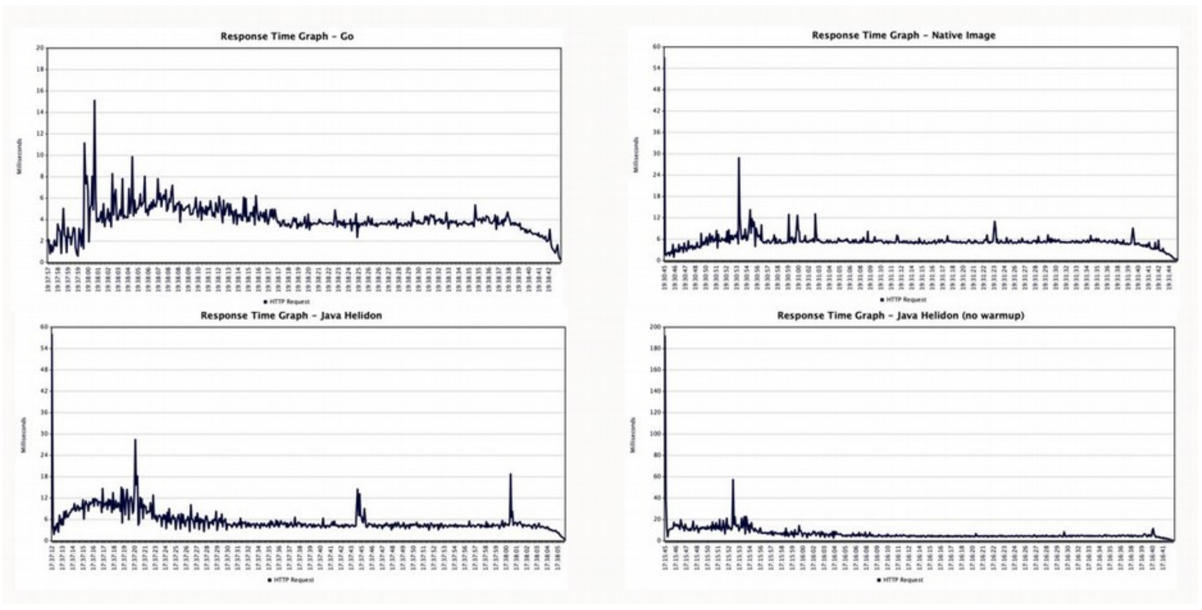
GraalVM has a native image feature that allows you to take a Java application and essentially compile it into natively executable code. From the GraalVM website:

> This executable includes the application classes, classes from its dependencies, runtime library classes, and statically linked native code from JDK. It does not run on the Java VM, but includes necessary components like memory management, thread scheduling, and so on from a different runtime system, called “Substrate VM”. Substrate VM is the name for the runtime components (like the deoptimizer, garbage collector, thread scheduling etc.).

Here are the round one results again with the GraalVM native image tests added (native images built with GraalVM EE 20.1.1 — JDK 11):

In this case we did not see any substantial improvement in throughput or response time by using GraalVM native image over running the application on a JVM, however the memory footprint was smaller.

Here are graphs of the response times over for some of the tests:



Notice that in all three Java variants the first requests had much higher response times (look for that blue line right up against the left axis). In all cases we also saw some peaks, which we assume to be caused by garbage collections or optimizations.

Round two

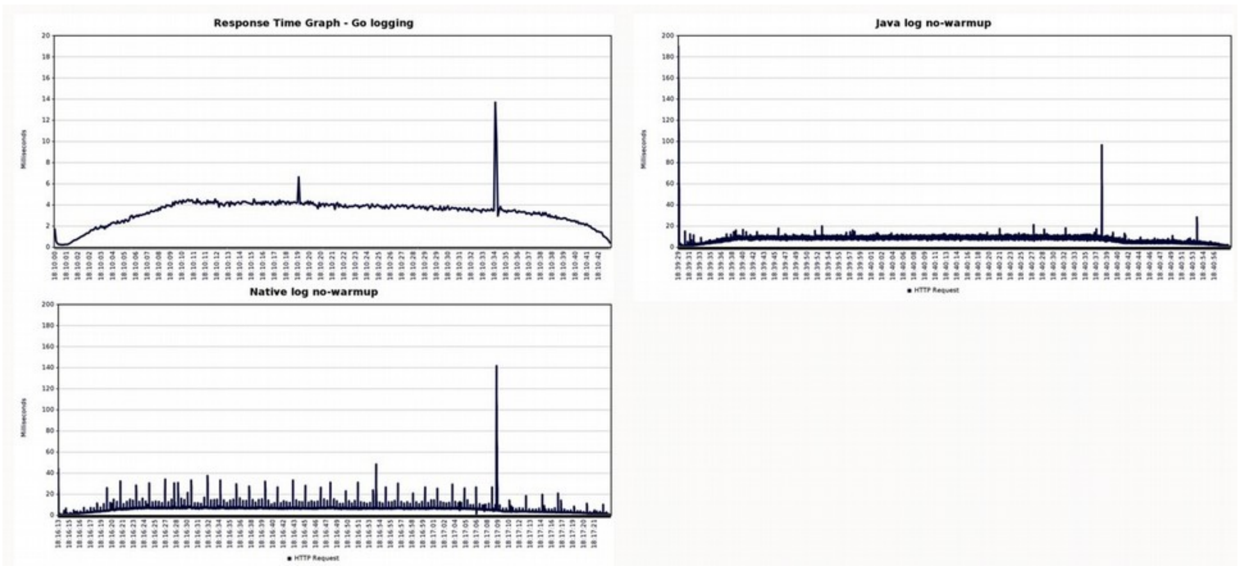
Next we decided to run the tests on a larger machine. For this round we used a machine with 36 cores (two threads per core), 256GB of RAM, running Oracle Linux 7.8.

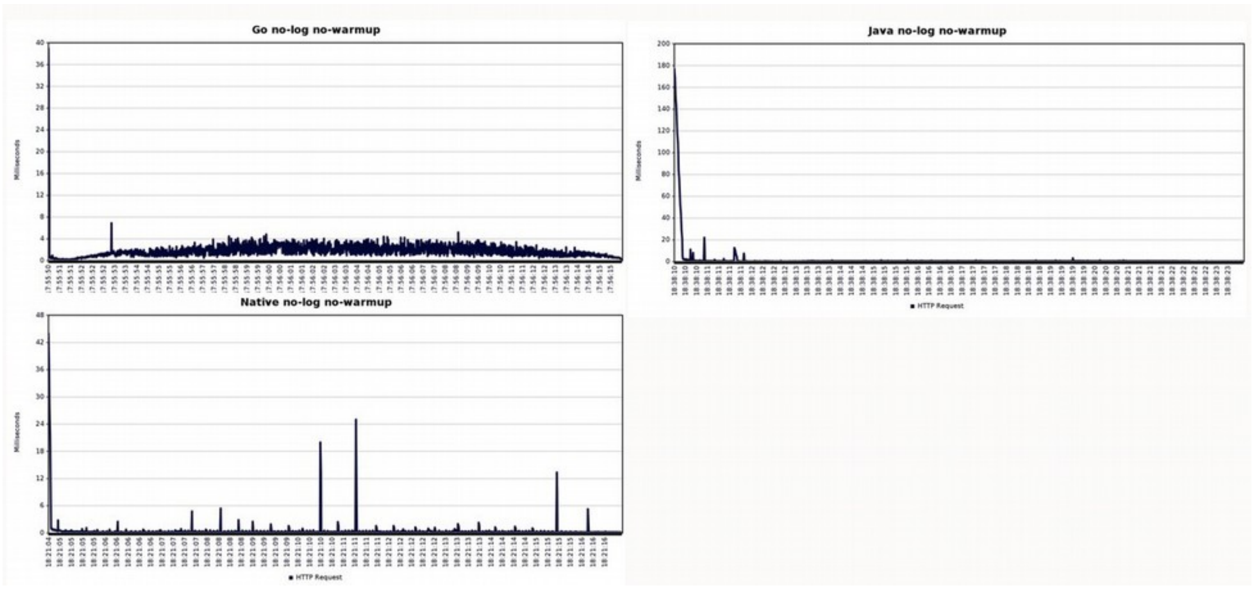
As in round one, we used 100 threads, 10,000 loops per thread, 10 second ramp up time and the same versions of Go, Java, Helidon and GraalVM.

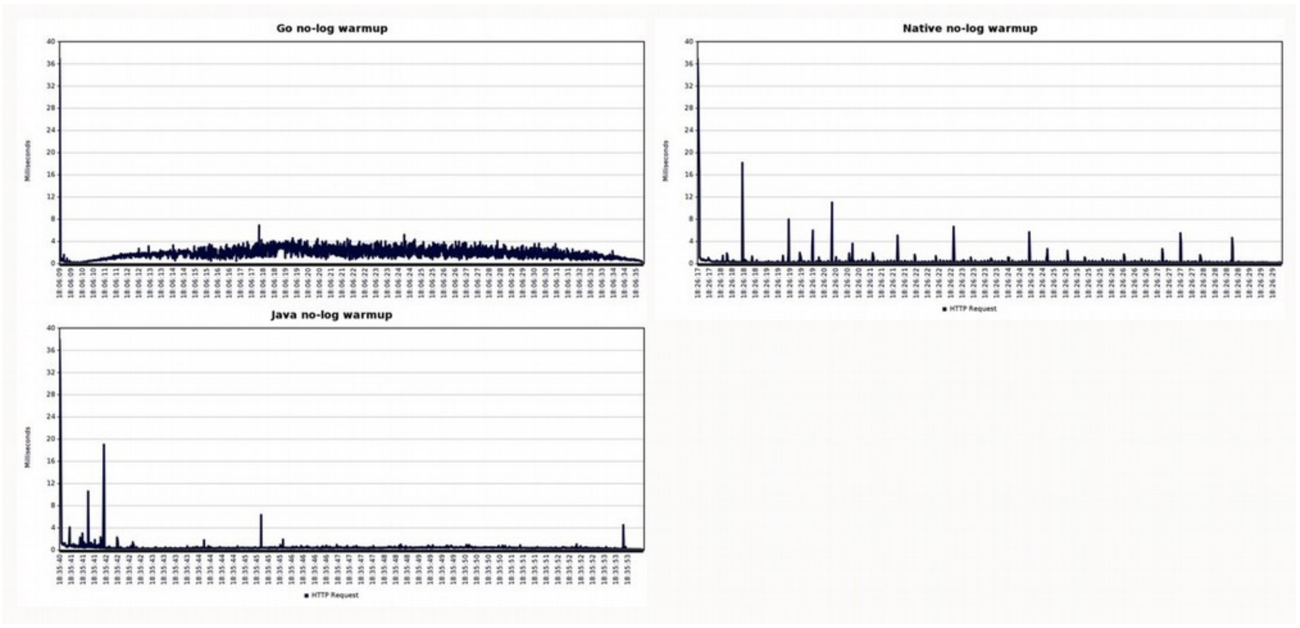
Here are the results:

We declared GraalVM native image the winner in round two!

And here are the response time graphs for these tests:







Some observations from round two:

- The Java variants performed much better in this test, and outperformed Go by a substantial margin when logging was not used
- Java seems to be much more able to use multiple cores and execution threads provided by the hardware (compared to Go) — this makes some sense because Go is intended as a system and network programming language, and it is a younger language, so it is reasonable to assume that Java has had a lot more time to develop and tune optimizations
- It is interesting to note that Java was designed at a time when multi-core processors were not commonplace, and Go was designed at a time when they were
- In particular, it appears that Java logging was successfully offloaded to other threads/cores and had a much smaller impact on performance
- The best performance in this round was from GraalVM native image, with a 0.25 ms average response time and 82,426 transactions per second, compared to Go's best result of 1.59 ms and 39,227 tps, however this was at the cost of two orders of magnitude more memory usage!
- GraalVM native image variants were around 30–40% faster than the same application running on the JVM
- The response time of the Java variants seemed to be much more consistent but with more peaks — we theorize this means that Go was doing more, smaller garbage collections

Round three — Kubernetes

In round three we decided to run the applications in a Kubernetes cluster — a more natural runtime environment for microservices you might say.

In this round we used a Kubernetes 1.16.8 cluster with three worker nodes, each one having two cores (with two threads of execution each), 14GB of RAM and Oracle Linux 7.8. We ran one pod for each variant in some tests, and one hundred in others.

Application access was through a Traefik ingress controller with JMeter running outside the Kubernetes cluster for some tests, and for others we used ClusterIP and ran JMeter in the cluster.

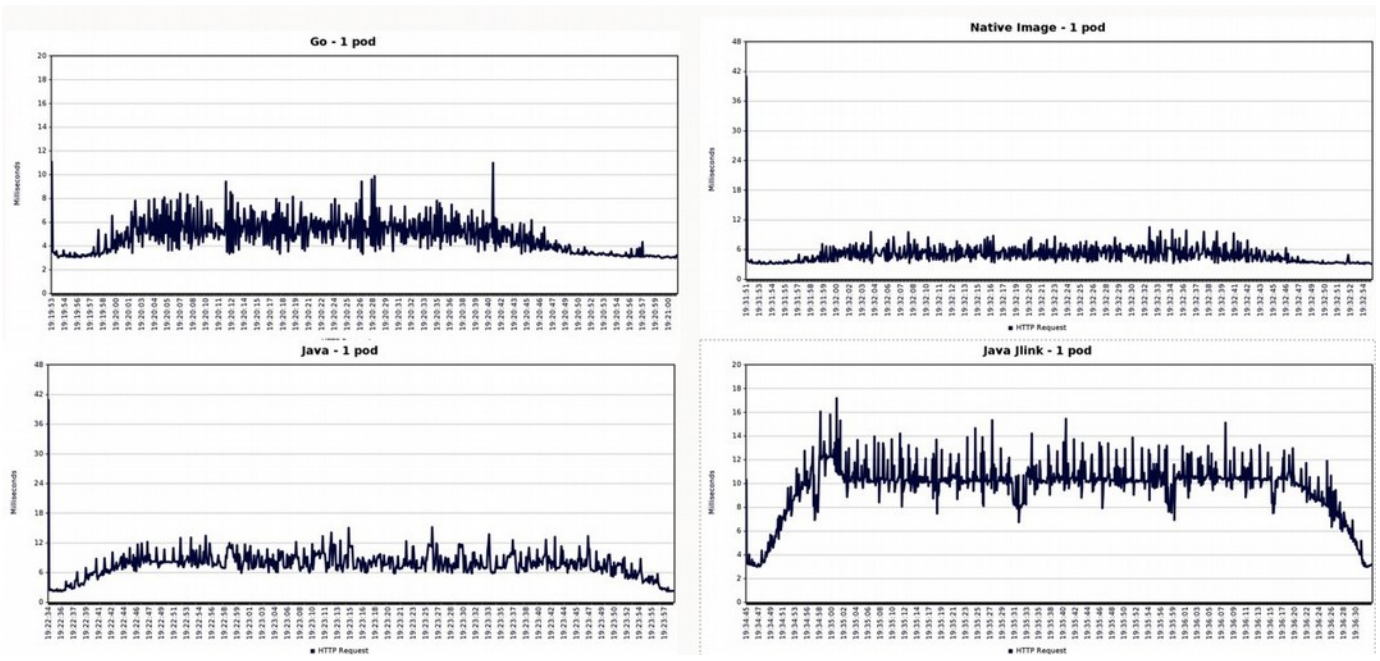
As in previous tests, we used 100 threads, 10,000 loops per thread and 10 second ramp up time.

Here are the sizes of the containers for each variant:

- Go 11.6MB
- Java/Helidon 1.41GB
- Java/Helidon JLinked 150MB
- Native image 25.2MB

Here are the results:

Here are some response time charts:



In this round we observed that Go was sometimes faster and GraalVM native images were sometimes faster, but the difference between these two was small (generally less than 5%).

So what did we learn?

We reflected on all of these tests and results, and here are some of our conclusions:

- Kubernetes does not seem to scale out quickly
- Java seems to be better at using all available cores/threads than Go — we saw much better CPU utilization during Java tests
- Java performance was better on machines with more cores and memory, Go performance was better on smaller/less powerful machines
- Go performance was overall slightly more consistent — probably due to Java’s garbage collection
- On a “production-sized” machine Java was easily as fast as Go, or faster
- Logging seemed to be the main bottleneck we encountered in both Go and Java
- Modern versions of Java, and new frameworks like Helidon, are making large strides in removing/reducing the pain of some of Java’s well-known and long established issues (e.g. verbosity, GC performance, start up time, etc.)

What next?

This was a very interesting exercise and we intend to continue working on this, in particular:

- We want to do more work with Kubernetes auto-scaling — we might need a more complex microservice or a much higher load to see some differences in performance
- We want to look at more complex microservices, multiple services, and patterns like circuit breaking, etc., and see how the network impacts the performance, and how to tune a network of microservices
- Also want to look at the logging issue to see what can be done to remove that bottleneck
- We want to look at the object code and compare the actual instructions being executed and see if we can do some further optimization in the code paths
- We wondered if JMeter could generate enough load without itself becoming a bottleneck, but our testing indicated that was not a factor at all, it could easily keep up with both the Go and Java implementations
- Want to do more detailed measurements of container startup time, memory footprint, etc.

As we do more experiments, and get more results, we will publish them here! Stay tuned and thanks for reading!