Revisiting Clojure

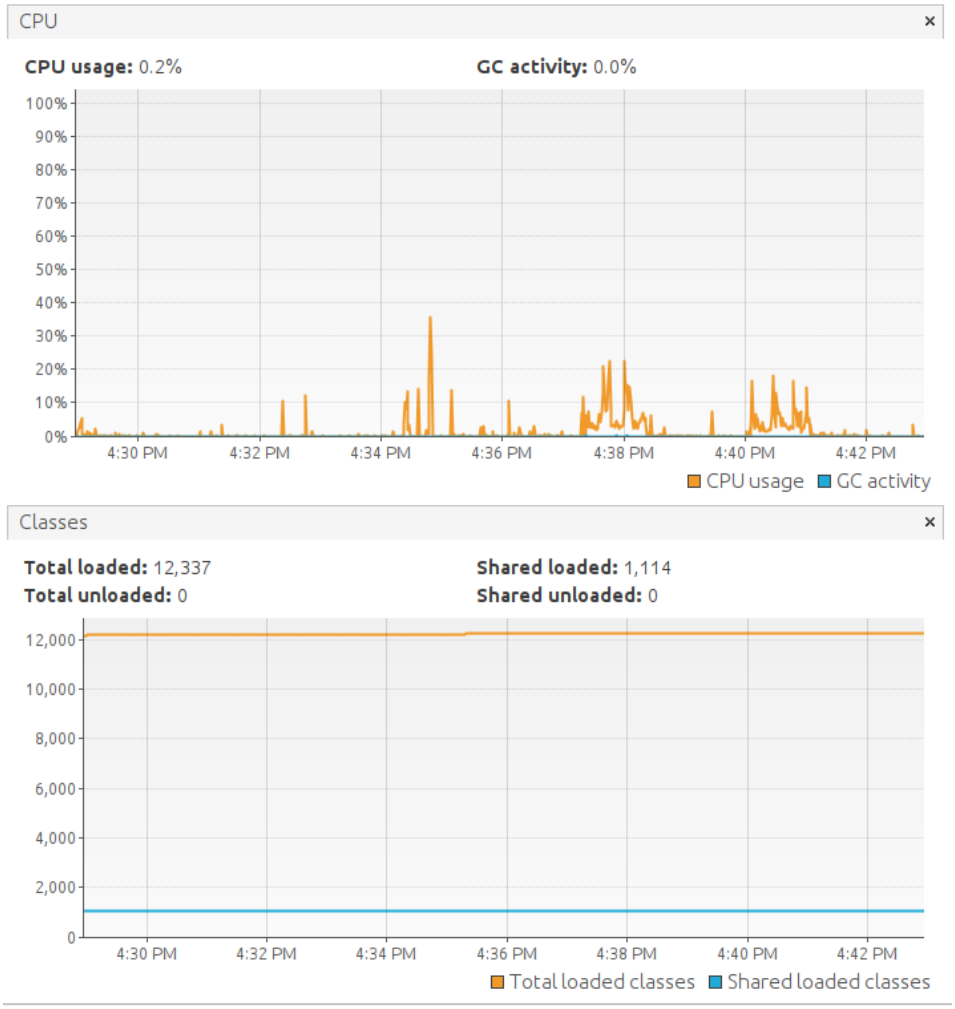
by Glenn Engstrand

In 2014, I started what eventually became a series of blogs where I implemented a feature identical microservice in various programming languages and technology stacks then compared and contrasted each implementation with the others in terms of architecture, design, coding, and performance under load. The first such implementation was written in Clojure on Ring and Jetty. When compared to what was to come, this first microservice turned out to be somewhat unimpressive.

I must confess to a nostalgic bias towards Clojure most likely because its predecessor Lisp really blew my mind back at university. I have been known to re-implement microservices in Scala four times already in order to improve Scala's standing. Late last year, I enhanced the Python on Flask implementation to be hosted on uWSGI in order to double the throughput and half the latency. I have been keeping my eyes open for any promising new Clojure stacks to evaluate.

I first learned about an open source project called Pedestal over a year ago. My original enthusiasm for this microservice framework for Clojure quickly faded when I realized that its routing architecture of interceptors, context binding, chain providers, and network connectors, though highly pluggable, was too complicated for serious consideration. I started with their basic template but even the slightest changes to the code would mysteriously break the service.

In January of this year, I ran across an open source project called Donkey in an InfoQ article. Donkey makes it easy for Clojure developers to write micro-services on the Vert.x framework. I had previously evaluated Scala on Vert.x and found it to be promising. Even though Donkey is only six months old and has but a single contributor, I decided to evaluate it anyway. A quick examination of its repo on github reveals that Donkey is, as of the time of this writing, composed of 26 Clojure files and 113 Java files totaling 12,194 lines of code (LoC).



Architecture

In order for each microservice to be feature identical and therefore comparable, the architecture has to be very similar. Like most of its predecessors, this new Donkey implementation (feed 13) is polyglot persistent with participants and friends in MySql fronted by Redis and news feed items in Cassandra made searchable by Elastic Search.

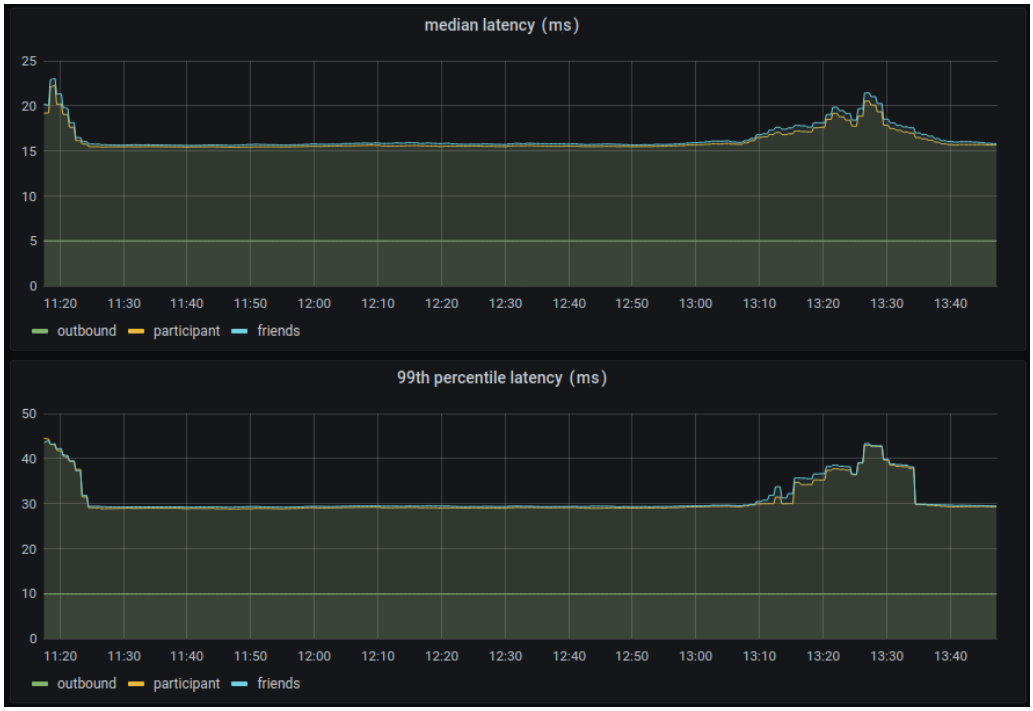
This implementation is like the previous two implementations (feed 11 and 12) in that the framework is reactive and the creation of outbound news feed items is asynchronous. I like how easy Donkey lets you handle that with a handler-mode attribute in the routing specification of blocking or non-blocking. Blocking handlers should return the result whereas non blocking handlers should pass the result into a lambda.

In Clojure, the def special form is evaluated at both compile and run time.

Perhaps the biggest architectural difference between that original implementation (feed 1) in Clojure and this one is in terms of the threading model. Feed 1 uses Ring which sits on top of Jetty which dedicates a thread for each inbound request. Feed 13 uses Vert.x which sits on top of Netty which uses the NIO library where there is no such per request thread affinity. What does that really mean to the application developer? You cannot perform blocking IO in the thread in which your code gets called by the framework.

Another big difference is that feed 1 is built by and runs on Java 8 whereas feed 13 is built by and runs on Java 11 due to the Donkey and Vert.x dependency requirements. Java 8 vs Java 11 has minimal impact from a Clojure developer perspective, at least in terms of coding. I used the Eclipse plugin Counterclockwise back when I was developing feed 1. It looks like that IDE doesn't work with Java 11. I get the feeling that Counterclockwise is more-or-less abandoned at this point. This blog is supposed to be focused on open source technology and I didn't really want to cover any of the proprietary IDEs so I just ended up using Emacs.

Here are some requirements in the first two feeds that got dropped by the third. The ability to switch out some backends (PostGres or MySql, Solr or Elastic Search, Redis or Memcached) based on a runtime configuration switch got dropped. The instrumentation of the microservice for monitoring purposes by sending performance data to Kafka got dropped. Feed 1 has all of these features whereas feed 13 has none of them.



Design

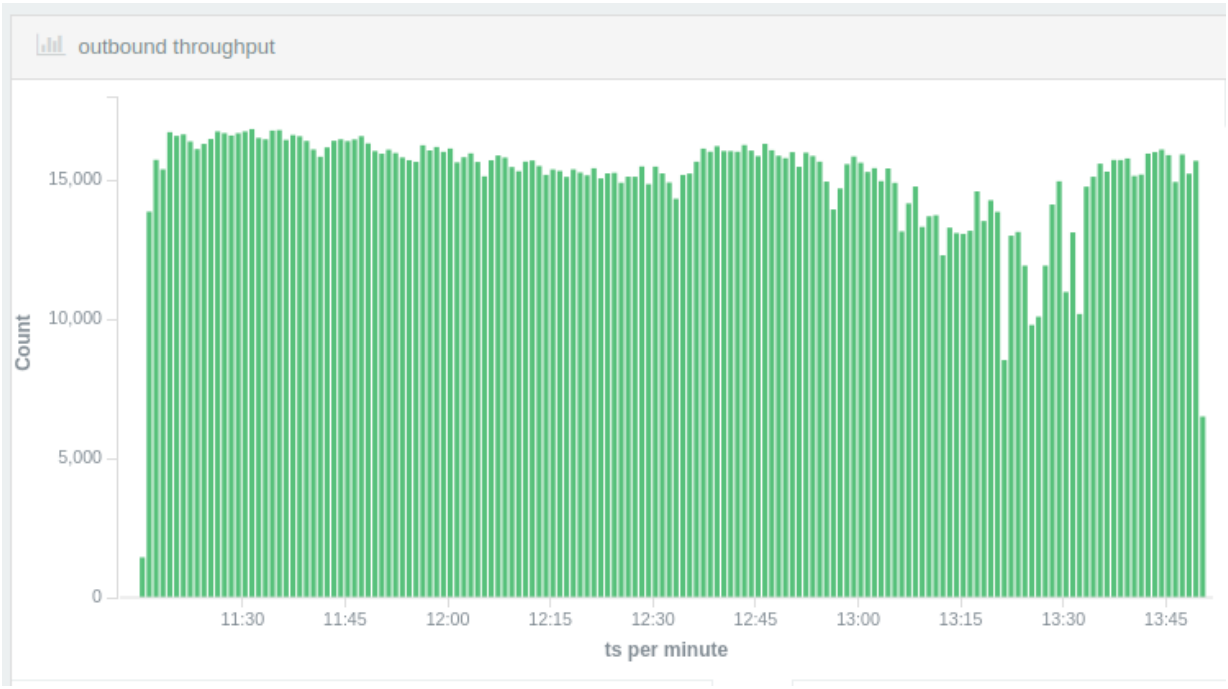
Feed 1 was a bit of a learning adventure for me as I was new to Clojure at that time. Feed 13 is designed in a more layered, modular, and modern way with controllers, services, and Data Access Objects or DAOs. When creating participants, friends, or outbound news feed items, feed 1 callers had to use the form post part of HTTP. Feed 13 just uses content type application/json request bodies.

Donkey claims to support Compojure routing which was used in feed 1 but I was not able to get it to work with query string parameters and request bodies so I dropped the use of Compojure in feed 13.

Here is a fun fact that you may not know about Clojure. The def special form is evaluated at both compile and run time. This is how you create global variables such as the singleton connections to the various underlying data stores. You may not have access to those data stores at compile time. The feed 1 code attempted to guard against connection initialization at compile time but in a way that was not guaranteed. The feed 13 code never attempts to connect at compile time. The code explicitly does this at runtime during service initialization.

I cannot find a good open source IDE for Clojure.

Leiningen (or lein) is Clojure's most popular build tool and it supports the ability to create uber or standalone jars with AoT (Ahead of Time) compilation. For reasons not entirely clear to me, Lein stopped being able to build the uber jar for feed 1. This made the docker build more complicated because it had to recreate the build environment in order to run it without the uber jar. Lein is able to build the uber jar in feed 13 so the docker build process is much simpler.



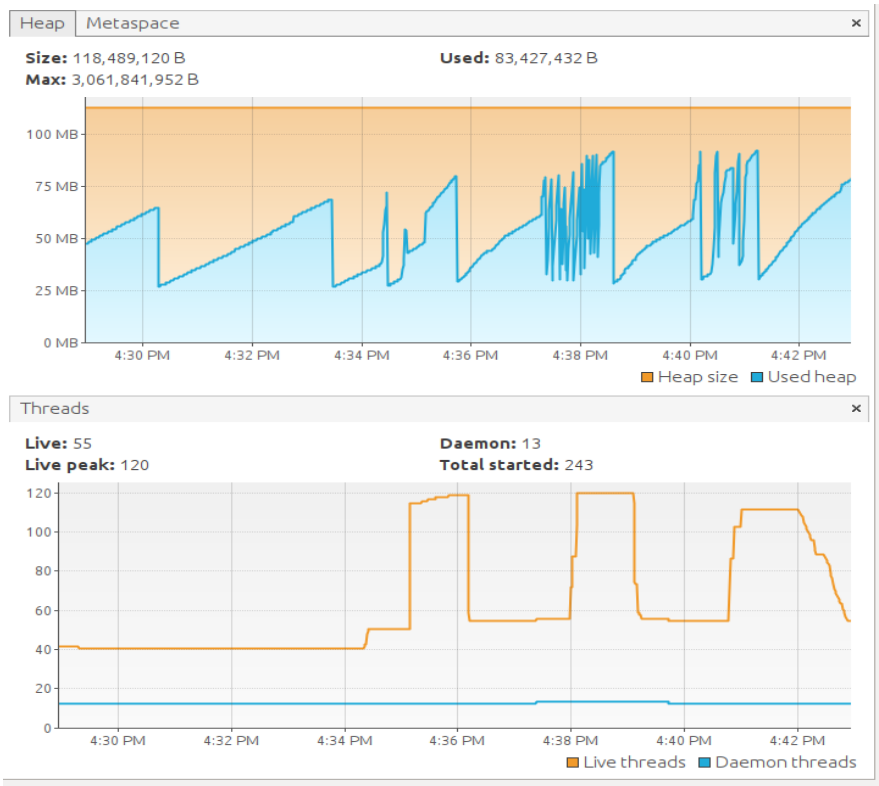
Code

The code base for feed 13 consists of 22 files totaling 593 LoC. The average file size is 34 LoC.

After studying the code for feed 1 and 13, you may notice some things missing in the new implementation. Feed 1 used records whereas feed 13 just uses hash maps. Feed 1 uses multi methods which are not in feed 13. Feed 1 recorded performance data as JMX metrics. This is missing in feed 13 because performance data is now collected by a custom proxy that publishes that data to both Elastic Search and Prometheus.

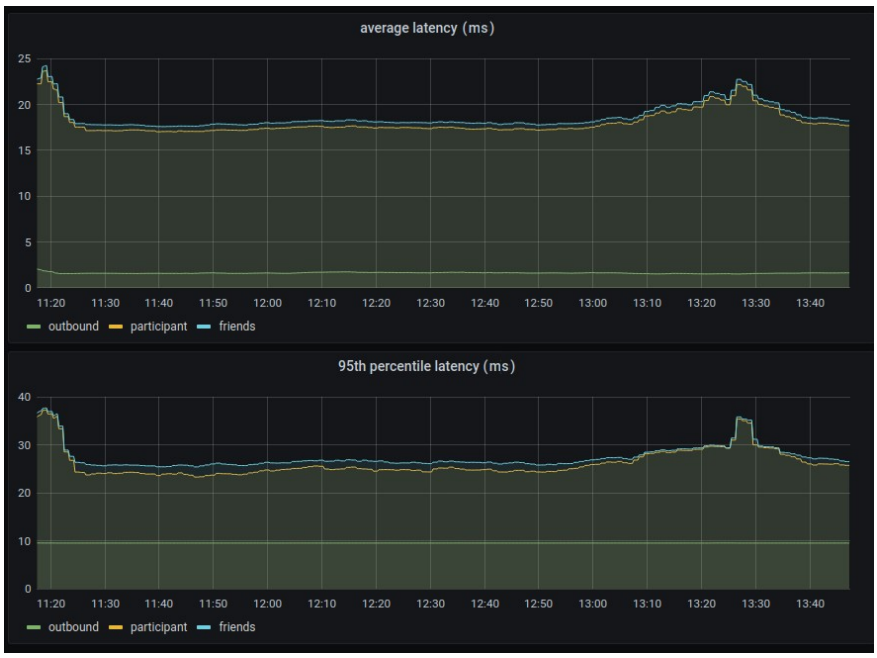
Feed 1 uses Clojure libraries for connecting to all the underlying data stores. Feed 13 uses the corresponding Java libraries instead. The Java libraries have more support and are kept up-to-date more frequently. This makes the DAO code harder to read since there is a lot of Clojure to Java interop. JDBI 3 was used for accessing the MySql database.

As mentioned earlier, the def special form is how global variables are defined and initialized. The connectors to the underlying data stores are global variables but they get initialized later at service startup time. This is accomplished in Feed 13 using a mechanism in Clojure for managing shared, synchronous, independent state known as atoms.



Test Automation

The unit tests for feed 1 are notoriously poor in terms of code coverage. This is more a reflection on my lack of commitment to unit testing in 2014 and some poor design choices I made in the day then it is a problem with Ring.

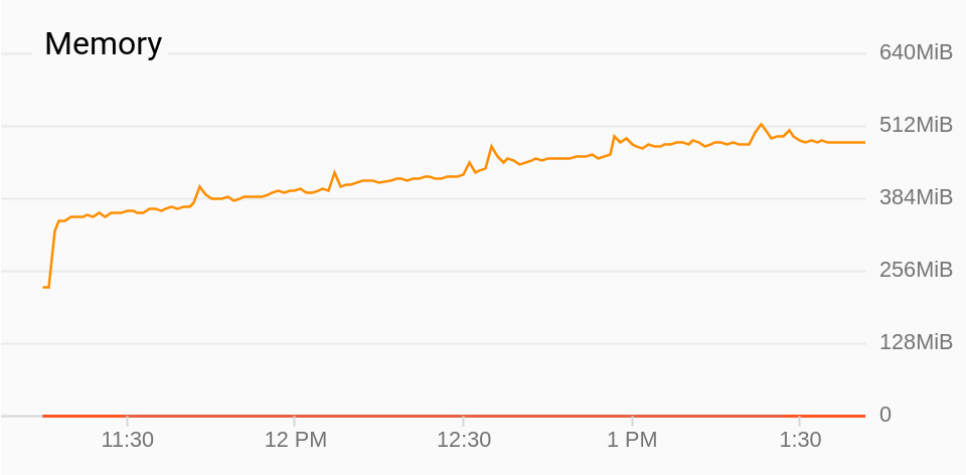


In feed 13, the unit tests cover a lot of the service code while mocking all of the DAOs.

The client load app has an integration test mode which I use to vet all of the endpoints while deving. Feed 12 has a Gatling integration which I use with Visual VM to profile feed 13. With a peak load of 16 create outbound requests per second (RPS), 4 create participant RPS, and 9 friend participants RPS, the Java Virtual Machine (JVM) showed brief spikes of up to 37% CPU, a peak of 120 threads, and up to 83 MB heap memory of which 40% was filled up with java.lang.reflect.Method objects.

Why so many reflection objects? As noted earlier, Clojure runs in the JVM. The Java programming language is statically typed but Clojure is not. That means that Clojure code has to use the Java reflection API in order to call Java methods. You can mitigate that with what is known as type hints. I don't use type hints in Clojure because I feel that they reduce the readability of the Clojure code. The consensus wisdom is to “avoid the use of type hints until there is a known performance bottleneck.”

That client load app can also be run as a load test. I ran the test for over 2 hours. There was a glitch in both throughput and latency near the end of the run but it recovered automatically without any pods restarting. CPU remained between 25 and 38 percent but RAM usage slowly yet steadily rose for the duration of the test peaking at about 0.5 GiB. Here is my guess as to the cause. The blocking IO part of create outbound is wrapped in a future which is backed by a Java cached thread pool in such a way that the number of threads that can be created is without bounds. Each new thread takes up some memory.



MySql performance was not the best. I chose JDBI because it performed so well in feed 3 so I do not believe that it is the cause of the poor performance here. The default pool size was 1 so I ran another test with a pool size of 18. This is the pool size used for feed 3. Feed 1 also used a connection pool.

There was no significant change in latency but the differences in throughput were quite extraordinary. The throughput increased by a third for both participants and friends (both of which always insert a row into MySql) but decreased by a third for posting outbound news feed items (which asynchronously queries MySql very infrequently). Something similar to this happened for feed 12.

Conclusion

Should Clojure developers choose Donkey on Vert.x or Ring on Jetty? Feed 13 certainly had less complex code than feed 1 with 18% less code overall and 39% less code per file. I could most probably achieve similar results if I rewrote the Ring on Jetty service with a better design.

Create participant or friend participants latency for feed 13 was 2 to 3 times slower than feed 1 but create outbound latency for feed 13 was 6 times faster than feed 1. Throughput for create participant or friend participants was about the same between feed 13 and feed 1. Create outbound throughput for feed 1 was 60% that of feed 13.

Any comparison of create outbound performance is unfair since create outbound is reactive on feed 13 but not on feed 1. In other words, the call returns for feed 13 before the work is done. For feed 1, the call does not return until all the work has completed.

I would ask that Clojure developers consider Donkey on Vert.x only if they want their microservices to be reactive and if they would be okay with the possibility that they might have to assume the responsibility of supporting that Donkey project at some point in the future.