

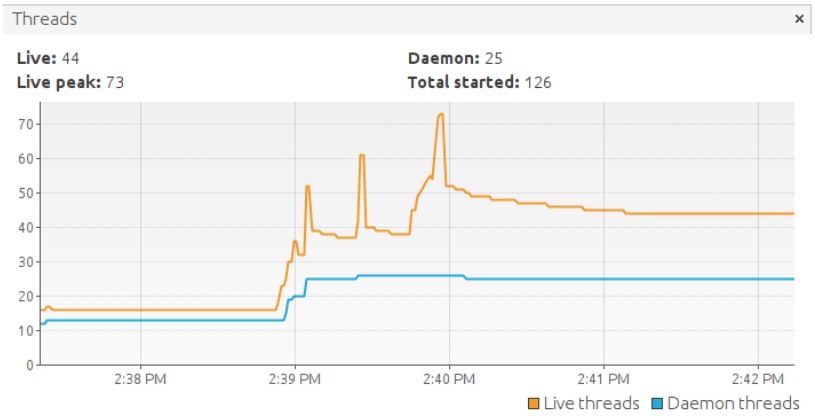
Writing Reactive Microservices for the JVM

by Glenn Engstrand

As modern software applications become more distributed, there is a growing trend where microservices become more reactive both in their internal structure and in the design of their APIs. In this blog, I compare and contrast two different reactive frameworks for the JVM; Vert.x and Play. For reference, I also compare these reactive frameworks with a servlet based framework. If you are interested in learning more about both the developer experience and performance under load for these reactive frameworks, then read on.

Originally developed at VMware and now under the guidance of the Eclipse Foundation, the Vert.x framework permits both Java and Scala developers to organize microservices into verticles each of which encapsulates a technical functional unit for processing events. The server backend for Vert.x is Netty.

Originally developed at Typesafe who later rebranded themselves as Lightbend, the Play framework also supports both Java and Scala developers. It can be configured to work on top of two different backend server frameworks, Akka HTTP and Netty.



Feed 12 (Play) thread count while Java profiling

Play runs hotter whereas Vert.x runs more smoothly.

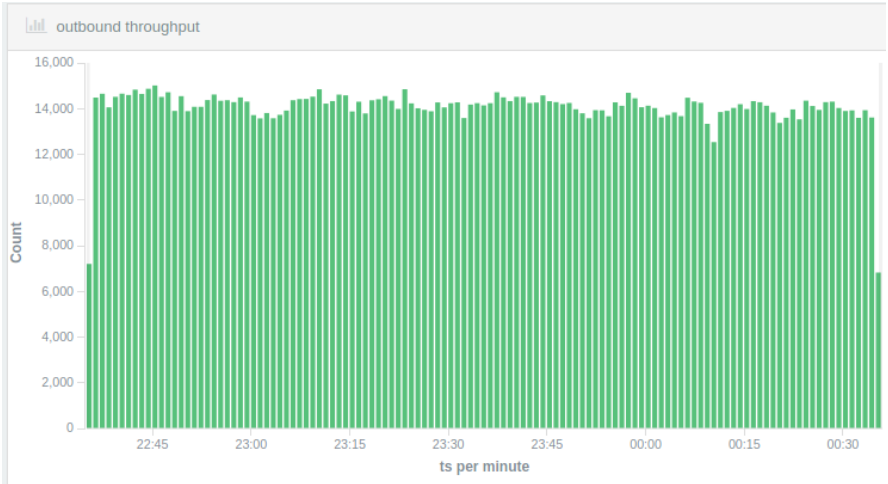
In terms of search engine popularity, Vert.x has steadily yet slowly climbed since it was first released by Tim Fox in 2011. Play was introduced in 2007 and peaked in 2014 but interest has declined to roughly at the same levels as Vert.x is currently. Both technologies are hosted on github which reveals that Play's community appears to be about an order of magnitude bigger and more active than the Vert.x community. At the time of this evaluation, the rate of commits to Play have dropped sharply since their April 2020 layoffs whereas the rate of commits to Vert.x have remained about the same.

I have this public github repo where I implement the same feature identical, polyglot persistent, rudimentary news feed microservice in different programming languages and frameworks. I run each microservice on the same test lab then capture and analyze the performance results in order to form a basis for comparison between these various programming languages. I followed this same pattern when I evaluated the Vert.x (feed 11) and the Play (feed 12) implementations.

Architecture

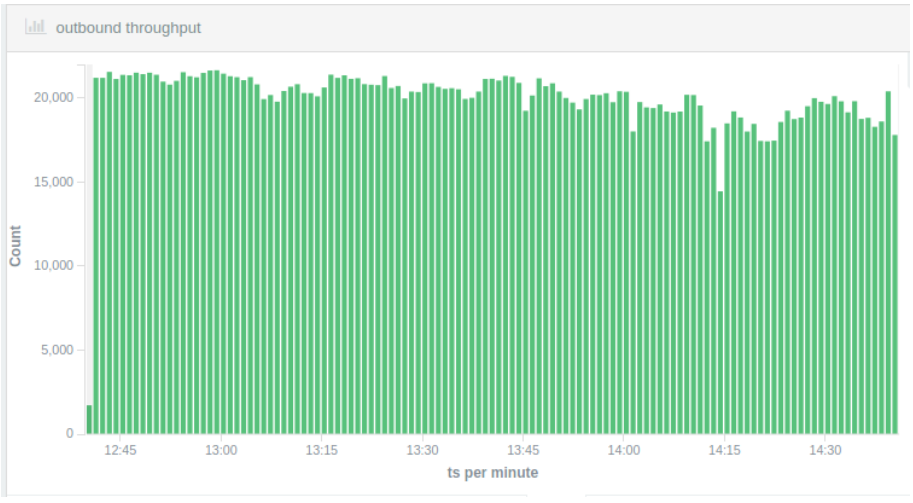
The architecture for both microservices is pretty much the same as the previous microservices (except for feed 7). They both use MySql fronted by Redis for participants and friends, Cassandra for inbound and outbound feed items, and Elasticsearch for keyword based search.

Feed 11 (Vert.x) average create outbound requests per minute

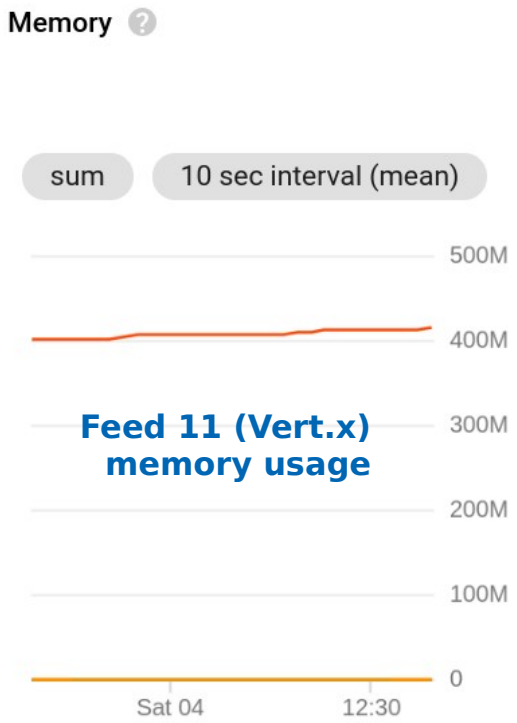


The biggest architectural difference between these two reactive microservices and their servlet based predecessors is the threading model. Servlet based frameworks must dedicate a thread for each request because either input or output operations normally block the thread until the operation completes. Both of these reactive frameworks can integrate with the Netty backend which uses the Java NIO library where channels, buffers, and selectors permit IO without blocking any application threads used to service inbound requests. Unlike the servlet based frameworks, there is no dedicated thread for each request so the service can accept a lot more connection requests than available threads.

When we say that a microservice is reactive, we could be referring to its internal structure, the design of the API endpoints it exposes, or both. Feed implementations 11 and 12 are internally structured reactively and surface endpoints that are a mixture of both traditional and reactive design. Since previous implementations of the news feed microservices are more traditional (every endpoint is synchronous except for inserting into Elasticsearch which is asynchronous except for the Clojure implementation) that makes many performance comparisons between these two new microservice implementations and their predecessors somewhat invalid. Here’s why.



Feed 12 (Play) average create outbound requests per minute



Feed 11 (Vert.x) memory usage

Traditional APIs are synchronous which means that every API call does what is asked of it before returning a response to the caller. In that way, it is said that these APIs are strictly consistent.

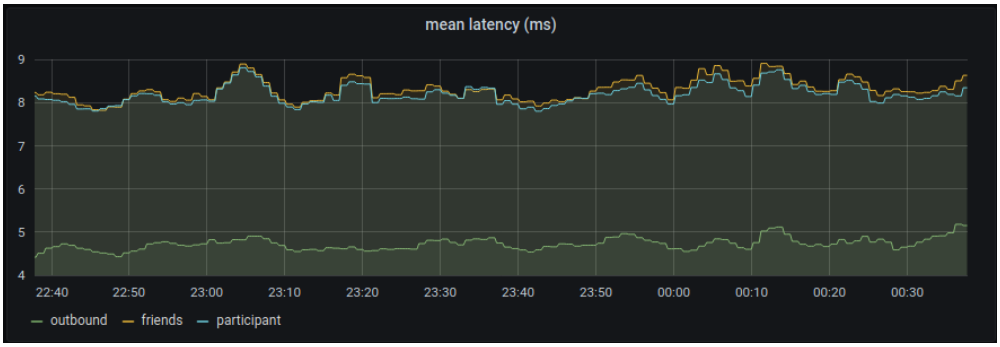
Reactive APIs are asynchronous. Calls to the API return a response before all the work is done. That makes these APIs eventually consistent. You can check to see if what you asked for is done immediately afterwards and it might not be done yet but it will be eventually.

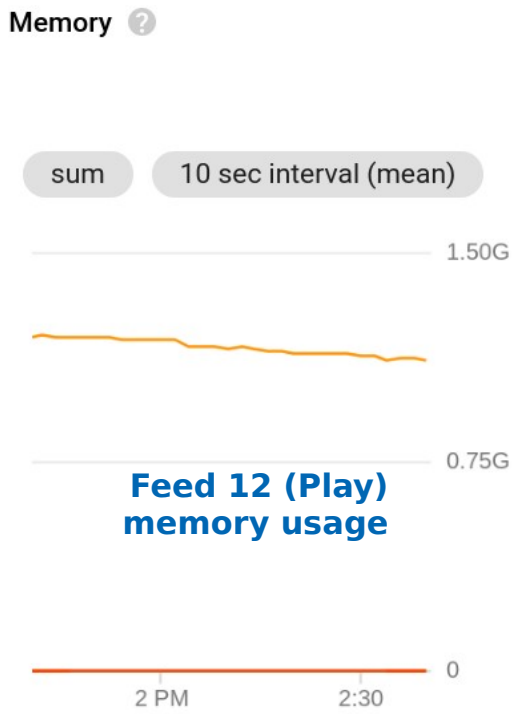
For both of these microservices, the create participants and friends endpoints are synchronous because otherwise the load test won't work properly. Creating an outbound post is asynchronous. That transaction is what we focus on primarily when evaluating performance during the load test.

Design

While the architectures are similar, the designs of the feed 11 and 12 microservices are quite different from each other.

Feed 11 (Vert.x) average latency





Feed 12 (Play) memory usage

As the Vert.x framework starts up, the main verticle defines all of the HTTP request routing and deploys the other nine verticles that service the event bus. Each handler for a request route puts the routing context (both request and response objects) on an internal cache then publishes the key to that routing context to the corresponding topic on the event bus for that particular type of request. An instance of the related verticle consumes the message, gets the routing context off of the internal cache, then processes the request. The response for the result gets returned once the end method on the response object is called.

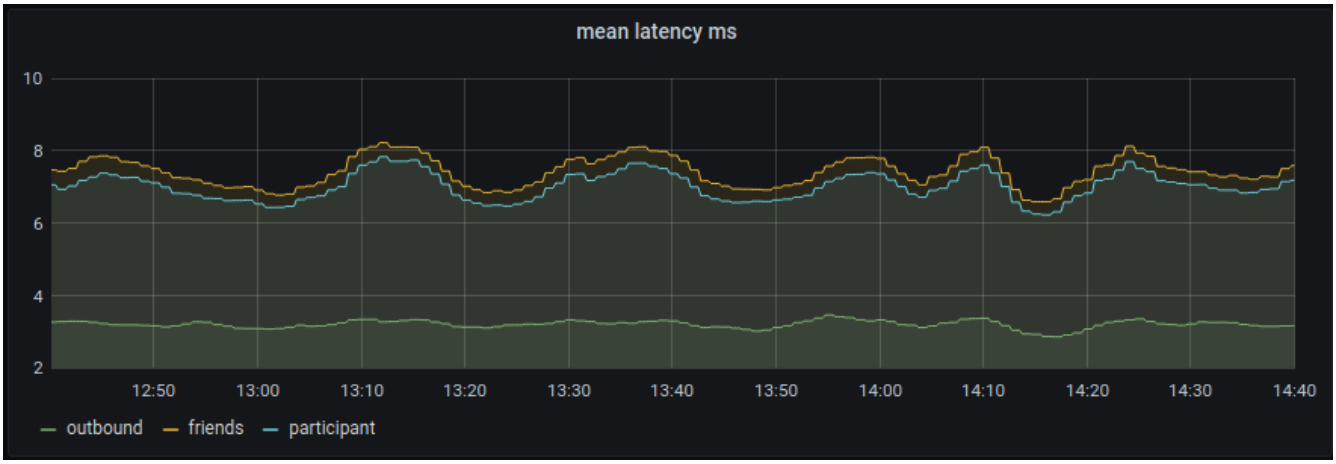
The biggest design decision with regards to parallelism is how many verticle instances per event bus topic and the size of the thread pool used by Slick (more on that later).

As the Play framework starts up, the routes file under the conf directory provides the mapping between the leftmost part of the path of each request to the appropriate sird router. Each router maps the remaining part of the path and the method to the appropriate method in the corresponding controller. The controller method, in turn, calls the appropriate service method. The controller also integrates with the Play action builder in order to deliver a Future[Result] back to the framework which, in turn, waits for the future to complete in a Netty compatible way in order to return the response.

The biggest design decision regarding parallelism here is how many thread pools (including Slick) and the sizing of each pool.

The servlet based news feed implementation that I will be comparing these two projects with is feed 6 running Scalatra which uses Jetty as the server backend. The routing is very DSL oriented and returns case class objects from the model package that get serialized via implicit type conversion with the json4s jackson library.

Feed 12 (Play) average latency



Coding

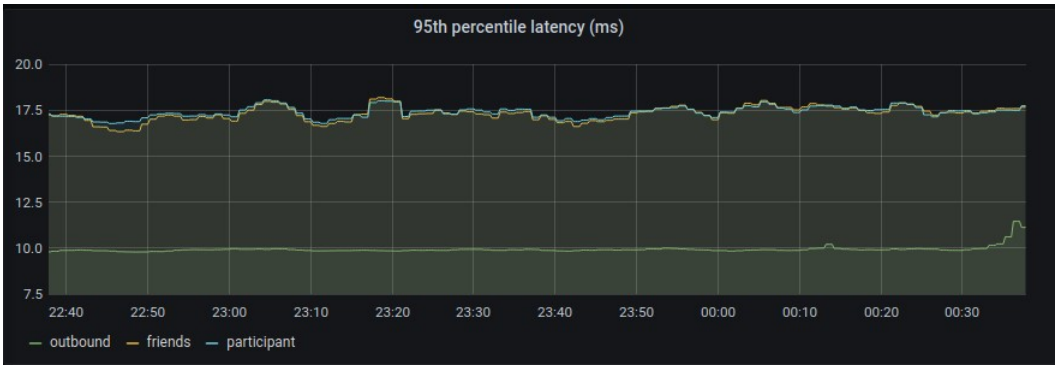
There is a lot of commonality in the Scala code for both feed 11 and feed 12 microservices. They use the same client libraries for accessing Cassandra, Elasticsearch, and Redis. They both use the Slick library for accessing MySql. The package structure is similar; resources, services, DAOs, and models. The feed 6 service uses earlier versions of the same client libraries and Doobie instead of Slick.

Both reactive microservices make use of the same popular Scala language features; for comprehensions, case classes, type classes, extension methods, and monads. Where practical, they both use Circe which is a Scala library for processing JSON. They both use Scala Test which is a framework for writing and running test automation.

There are a lot of differences in the code too. The Vert.x microservice also has an events package which holds all of the event bus consuming verticle code. It also integrates with ehcache as the internal cache for the routing context objects. You cannot put routing context objects directly on the event bus because they cannot be serialized. Why can only serialized objects be published? Because the Vert.x event bus can be distributed via a configuration switch with an assortment of cluster managers that integrate with such technologies as Apache Ignite, Apache Zookeeper, and Hazelcast.

Due to how the Play framework operates, services and DAOs need only return results wrapped in futures. Because of that, there is no need to explicitly code for a message bus. You have to be careful to never wait on any future or perform blocking IO without being wrapped in a future or your whole microservice may become unresponsive.

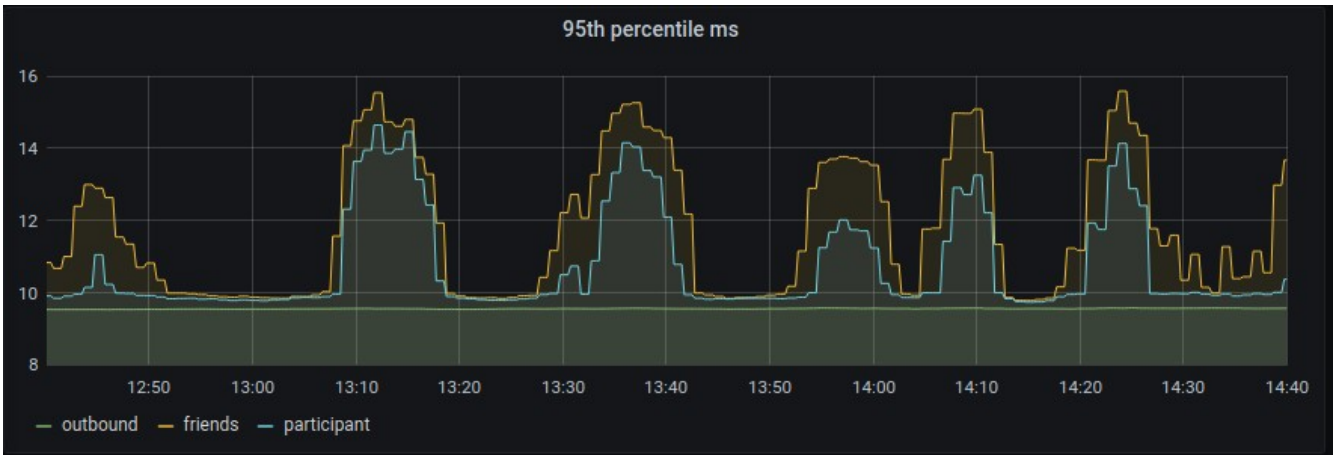
That is the biggest difference between the two frameworks in terms of coding. With Play, your code returns the future wrapped response to the request. The code does a lot of future mapping in order to compose them. With Vert.x you pass that response into the end method of the response object. For that reason, the Vert.x code has a lot of lambdas but few return values.



Feed 11 (Vert.x) 95th percentile latency

There are three thread pools for the Play service. Slick gets one pool. The DAOs, Cassandra, and Elasticsearch clients get a “repository.dispatcher” pool. Services and controllers use the default pool.

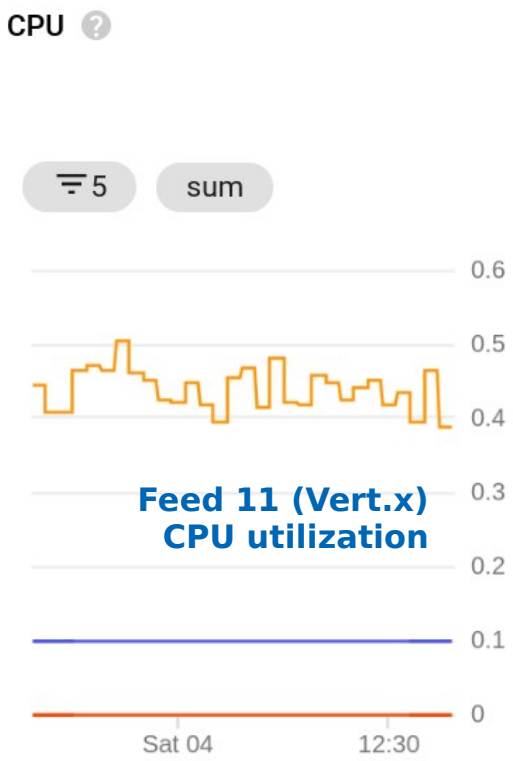
Feed 12 (Play) 95th percentile latency



There is one more coding difference between these two microservices, dependency injection. I just used Play's out-of-the-box integration with Guice whose module binds either real DAOs or mocked DAOs depending on the value of the environment.mode enum. I didn't even bother with proper dependency injection with the Vert.x microservice. The DAOs are traits with a real implementation and a mocked implementation. The service objects have mutable DAO fields that get initialized to the real classes but overwritten to the mocks in the unit tests.

Here is the static code analysis for these projects. For the Vert.x service, average per file Lines of Code is 42.33 with a median of 33 and a standard deviation of 33.58. The single unit test Scala file is the largest with 200 LoC. Total McCabe cyclomatic complexity is 1,300. This project is dependent on 136 external jars.

For the Play service, average per file LoC is 55.48 with a median of 51 and a standard deviation of 32.92. The news action builder Scala file (mostly boilerplate) is the largest with 123 LoC but the unit tests file is a close second at 122 LoC. Total McCabe cyclomatic complexity is 1,764. This project is dependent on 177 external jars 90, of which it has in common with feed 11.

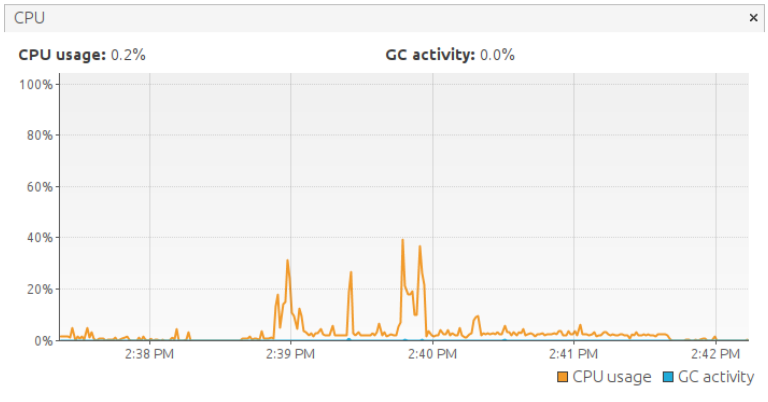


Feed 11 (Vert.x) CPU utilization

For the Scalatra service, average per file LoC is 40.15 with a median of 36 and a standard deviation of 29.47. Total McCabe cyclomatic complexity is 1,981. This project is dependent on 110 external jars.

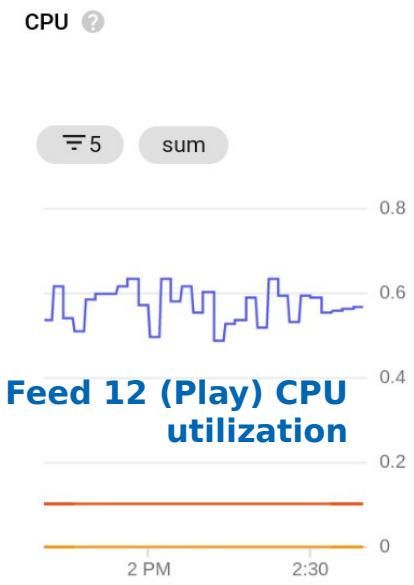
Testing

Unit testing for both reactive microservices exercises all code except for the DAOs. With Vert.x, the unit test code calls the createHttpClient method specifying localhost as the host. For Play, just create a FakeRequest object. There is a Gatling plugin for load testing via sbt which I used to profile the JVM running locally on my dev laptop.



Feed 12 (Play) Garbage Collection

I use a Kubernetes hosted load test environment when evaluating performance under load for these news feed microservices. In the past, I always used the create outbound news feed item API call as the basis for comparison because that endpoint does the most work. WIth these two microservices, that endpoint does the least work because now it is asynchronous. I still want to start with that endpoint because it is precisely that reactive nature that is the focus of this investigation.



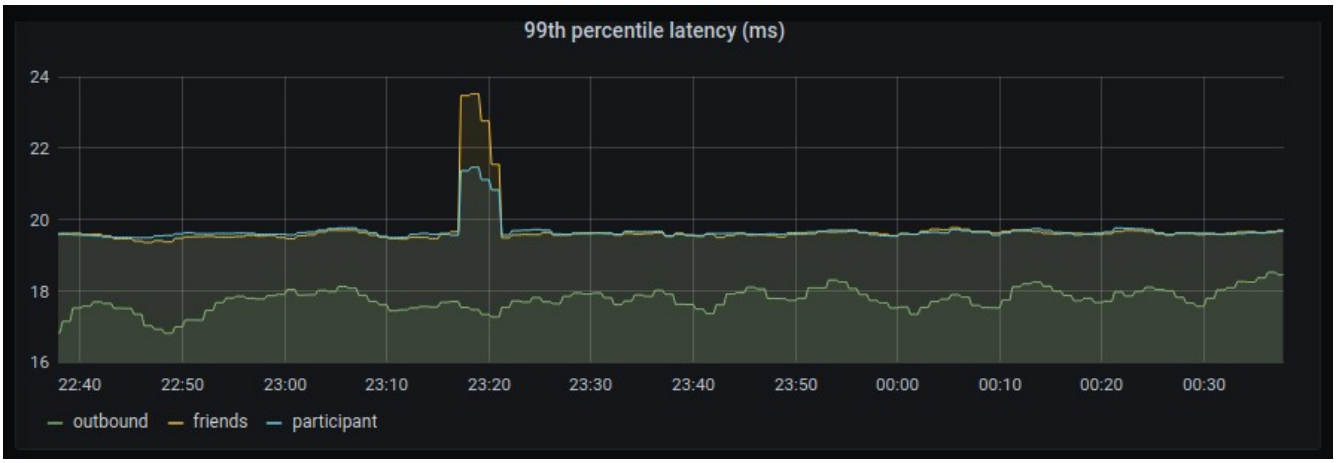
Here are the load test results for create outbound on the Vert.x microservice. Average per minute throughput of outbound posts is 14,136. Mean latency is 5.3 ms. Median latency is 5 ms, 95th percentile is 11 ms and 99th percentile is 15 ms.

I include here the load test results for create outbound on the Play microservice configured to run with both Akka HTTP and Netty as the server backends.

For Akka HTTP, average per minute throughput of outbound posts is 14,255. Mean latency is 3.3 ms. Median latency is 3 ms, 95th percentile is 6 ms and 99th percentile is 9 ms.

For Netty, average per minute throughput of outbound posts is 20,151. Mean latency is 4.4 ms. Median latency is 4 ms, 95th percentile is 8 ms and 99th percentile is 11 ms.

The situation is somewhat reversed for create participant which is a synchronous API that inserts a row into the participant table in MySql. For Vert.x, average per minute throughput is 3,035. Mean latency is 8 ms. Median latency is 7 ms, 95th percentile is 14 ms and 99th percentile is 19 ms. For Play, average per minute throughput is 1,824. Mean latency is 7 ms. Median latency is 7 ms, 95th percentile is 10 ms and 99th percentile is 13 ms. There would be periods of high latency at the 95th percentile for the Play service. Latency for the Vert.x service was more steady.



Feed 11 (Vert.x) 99th percentile latency

Since the create participant endpoint is synchronous, we can compare its performance metrics to previous news feed implementations. Average per minute throughput for feed 6 (Scalatra) is 3,885. Mean latency is 6 ms. Median latency is 6 ms, 95th percentile is 9 ms and 99th percentile is 12 ms.

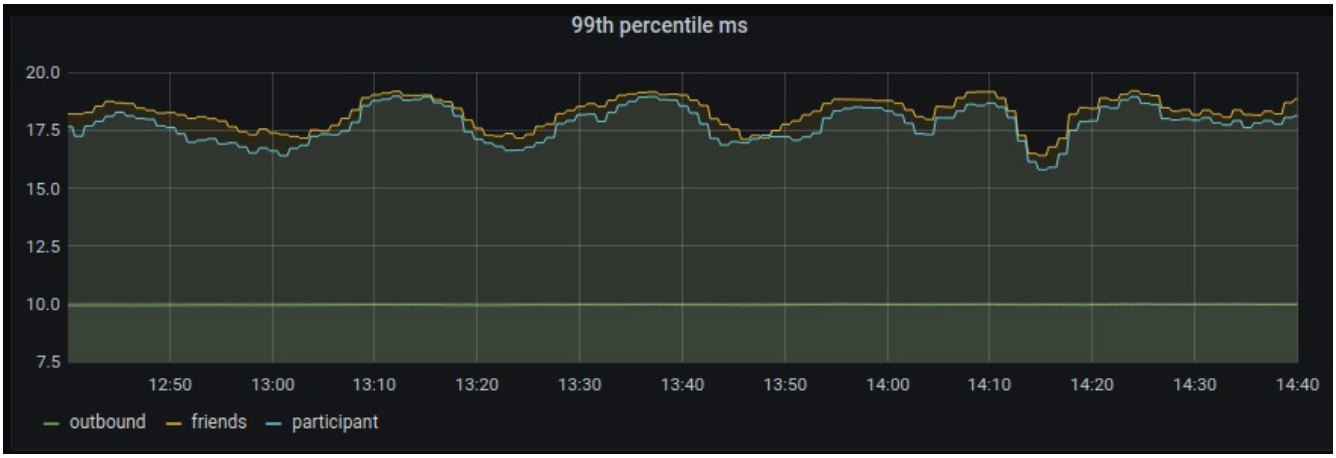
For the Play service, the “repository.dispatcher” thread pool is of size 20 and the Slick database connection pool has 10 threads. The service runs on a pod with no limits and on a VM with 4 vCPUs. The default thread pool size is matched to CPUs. Be advised that there are other thread pools in the JVM including those dedicated to Netty, Akka, Logback, and the JRE. I saw a peak of 73 threads when profiling with Gatling and Visual VM.

I ran the Play on Netty load test again after increasing the thread pools to 30 and 20 respectively. Average per minute throughput of outbound posts dropped to

14,803 but throughput for participant posts raised to 5,922.

I ran the Vert.x load test again after making the corresponding changes there. Average per minute throughput of outbound posts raised slightly to 14,949 and for creating participants raised slightly to 3,161.

Feed 12 (Play) 99th percentile latency



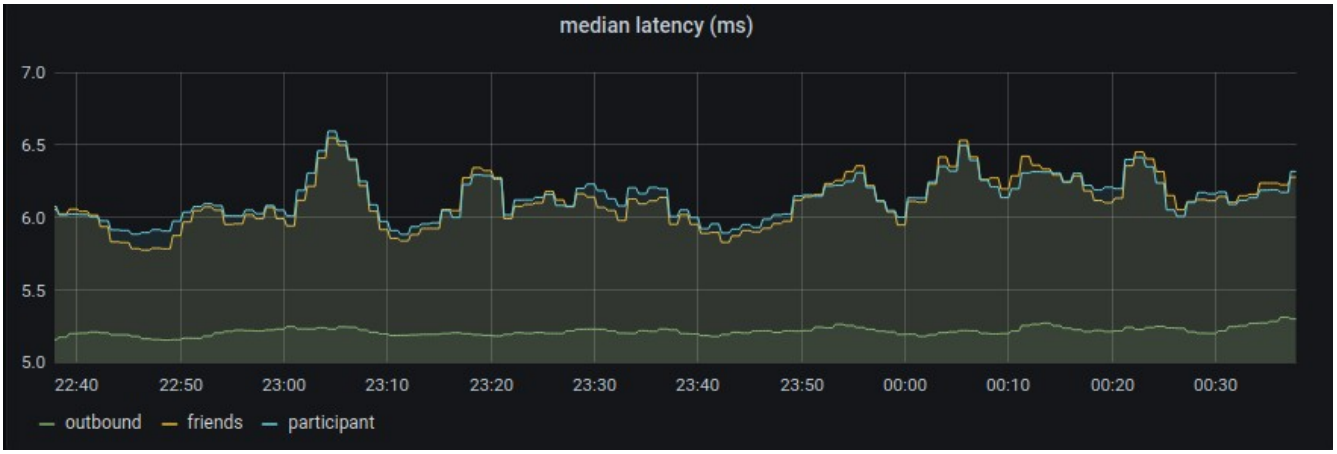
Resource utilization for all tests were very comparable, very stable and well within reasonable limits.

Conclusion

Just to be clear, I believe that both of these reactive frameworks are awesome and would feel comfortable basing the next application on either one of them. The same could be said for the servlet based predecessor too.

Play performs much hotter when configured to run with the Netty server backend than with the Akka HTTP backend.

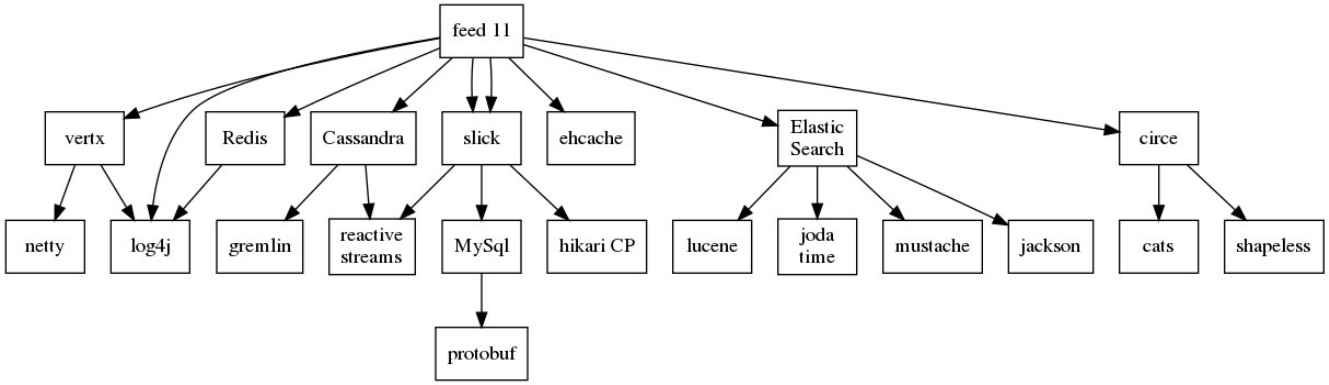
With enough thread tuning, Play apps can outperform Scalatra apps.

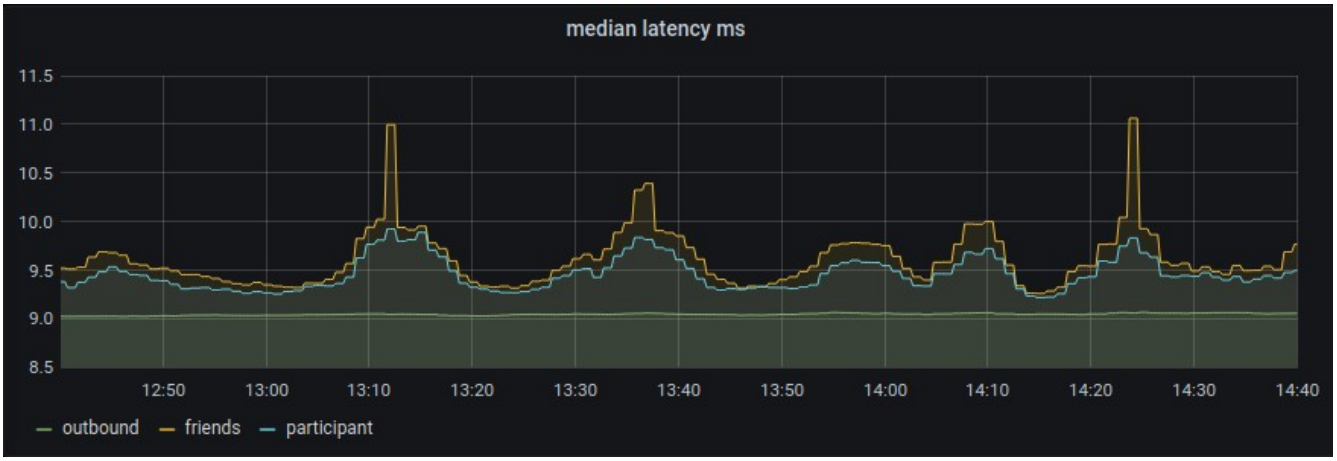


Feed 11 (Vert.x) median latency

The Play service was roughly a quarter more complex in its coding than the Vert.x service but with significantly better throughput for either asynchronous APIs or synchronous APIs but not both. Latency and resource utilization for the two microservices were in the same ballpark.

This could be a bias of mine since I am no stranger to pubsub but I feel like the explicit message management in Vert.x is a little easier to follow than the everything-is-a-future approach to Play. Perhaps I could have gotten better throughput with more thread tuning but that is a lot of knobs yielding fairly hard to predict results. I didn't feel like the thread tuning in Play was going to quickly converge to an optimal solution.





Feed 12 (Play) median latency

I could have added another layer of explicit message management such as the Actor ask pattern to the Play service but such an addition would have increased the code complexity of that service even more. It is unclear whether or not adding another layer on top of what is already causing the difficulty in thread tuning would have resolved that issue.

Perhaps you can get better performance out of Play than Scalatra but be prepared to spend some time tuning that threading model. Asynchronous APIs should perform better than their synchronous counterparts but at the cost that what was requested is not guaranteed to always happen.