Comparing Akka and Spring JMS

Mati Vait

***Abstract*—Reactive systems [1] need to be built in a certain way. Specific requirements that they have, call for certain kinds of programming techniques and frameworks. In this work we compare and benchmark two frameworks for building distributed concurrent applications from the Reactive Systems' point of view. We discuss each of the frameworks in depth and later reason about their shortcomings and strong-points.**

***Keywords*—Spring JMS, Akka, benchmarking, fault tolerance, high-availability, reactive systems.**

I. Introduction

REACTIVE systems [1] currently lie on the final frontier of the technology for processing large amounts of data. Various kinds of networks, both machine- and human-related, generate vast amounts of data that need to be processed in real-time. Also the data queries need to be responsive in a sub-second time-frame and scale both vertically and horizontally.

These requirements call for more flexible architecture and implementations that are de-centralized and that can be deployed or migrated from site-to-site on a moments notice.

Multiple technologies and software stacks are available for this purpose. The main goal of this paper is to investigate more closely the performance and suitability of two of those; a message passing framework - Spring JMS [2] and a concurrent event-driven actor-based framework - Akka [3].

Akka and Spring JMS were chosen because, as far as we know of, no such comparison has yet been done between two of these frameworks.

II. Technologies

Akka [3] and Spring JMS [2] were selected for both being able to abstract inter-application communication to intra-application communication. This means that neither for Akka nor Spring JMS it does not matter whether the receiving end of the communication is running in the same Java Virtual Machine (JVM) or not.

Our questions are how efficiently are they doing remote communication, how well do they perform under bigger loads and how do they fail, under what conditions, and how do they do recovery?

Next, we will discuss both technologies in more detail.

A. Akka

Akka is an actor-based technology for concurrent message-passing and ubiquitous application distribution. Applications can be written using either Java or Scala.

Akka was inspired by the ideas used for building Erlang[1][4] language. One of the main abstractions in Akka framework is the actor.

The actor [5] is already an old notion of abstraction describing concurrent systems part of functionality that is concerned only with one and only one aspect of computation and data transformation. Actors are connected to each other via abstraction called "mailboxes". Information is passed from one actor to another through mailboxes using the the specific actor reference. That reference is used internally in Akka to resolve the location of target actor.

Akka enables to send messages between actors[2] in one JVM as easily as between multiple JVM-s that are running in separate servers.

The distribution in Akka components is achieved only by using minimal changes in configuration file. Source code for the application can stay unchanged. In configuration files the developer needs to define the location of specific actor in terms of location it needs to be running in. Later, during the startup and application running the required actors are deployed on respective machines. For an example, code listing VIII enables to deploy actor "/remoteActor" on a certain machine running at IP 10.0.0.42 and port 2552.

Concurrency in Akka is built using real threads in the lowest layer called kernel. Akka's kernel does all of the fine-grain resource management and execution of actors. This causes some unpredictability to occur in the execution order and performance. Different JVM configuration, state and uptime may cause fluctuations in the performance of Akka-related components, some actors that are written serially in the code may be run in arbitrary order.

The regular JVM threads are still available to use in the actors, however that is not advised because it breaks the independence principles of Akka components. Also the scaling properties suffer when threads are used inside actors because it makes harder to scale (mainly) horizontally and also when that specific actor happens to be replicated in big numbers.

Fault tolerance in Akka is achieved by building hierarchies of actors using strong parent-child relationships. In this setting each actor may have one or more child-actors that are responsible for their own specific functionality. When failure occurs in one of the child-actors the failure is propagated upwards until it is handled either by restarting faulting subsystem, replacing that or by shutting down entire system. Akka adheres to Erlang's "Fail fast"[6] mantra.

B. Spring JMS

Spring JMS (Java Message Service) [2] is a messaging middleware. It is an implementation of Java JMS API. JAVA JMS API is a Java Message Oriented Middleware (MOM) API for multiprocess synchronous and asynchronous communication, that has been through many iterations of improvements.

[1] Erlang is a concurrent soft real-time language developed by Ericsson.

[2] In Erlang a process is an equivalent of Akka's actor.

Numerous enterprise-grade applications have been built using some kind JMS implementation as a messaging layer for distributing and scaling application's components. One of the most popular of those is Spring JMS implementation.

Spring JMS uses additional component called message broker for delivering the messages. In the context of this work, the ActiveMQ[7] is used with default settings.

The communication models are divided into two main categories:

1) Point-to-point: communication happens between strictly defined endpoints - a message is sent and delivered to specific receiver. This kind of channel between two points is called "queue".

2) Publish-subscribe: communication happens via "topic"-s. A "topic" is a channel that can connect many message producers to many message consumers. Messages of various topics are produced by various producers that "publish" the messages under various "topic"-s. Message consumers are subscribed to "topic"-s and receive messages published under respective topics.

Messages can be delivered either synchronously or asynchronously. The synchronous delivery blocks running thread until delivery ACK (acknowledgement) is received. Asynchronous send is performed in background thread and current running thread continues without blocking.

Receiving of the messages on the consumers are done via message listeners. Each listener runs usually in a separate thread.

Messages by JMS API specification are not required to be delivered in the order they were produced. The order can be altered by sending for example a message with higher priority after messages with lower priorities.

Fault tolerance is addressed in JMS as needed specifically inside the application code. JMS API only guarantees the message delivery if physically possible, otherwise the delivery error is thrown. Producer/consumer failures need to be handled on case-basis.

III. Benchmarking

A. Benchmarking framework

During the experimenting phase we tried out multiple benchmarking frameworks. Two of those got more attention - Caliper[8] and Java Microbenchmark Harness[9] (JMH). We decided to continue with the JMH because of the reasons described below. However, Caliper still has good potential.

1) Caliper: At first we started out with Caliper[8]. The annotation-based configuration and convention-over-configuration approach seemed to have good features and enough functionality to start with. But some pitfalls came up, that are mainly due to the active development. It's documentation and code base are not aligned. Missing classes, parametrization that's supposed to work but doesn't. But there is also a bigger problem - the benchmarking results are automatically uploaded to external server. This is an issue in it's own right - without any warning the output reads "Visit external url for seeing results". This functionality can be overridden but as of the time of experimenting, we were unable to set up a results server of our own because neither it's source code nor installation package were not available.

2) Java Microbenchmark Harness: Java Microbenchmark Harness[9] or JMH for short, is a framework built by Sun/Oracle performance engineers. It is an OpenJDK subproject that tries to address short-comings of Caliper framework.

Configuration-wise the JMH does not differ much from Caliper framework except for being more configurable output-wise and also more flexible parametrization and larger control over runtime parameters.

JMH is fully configurable from the code via Java annotations. It's only drawback is that it is currently not available via standard Maven[10] build-system repositories and has to be installed manually into local repository. However, after installation it can be used as any other Maven dependency. Compared to Caliper, JMH's documentation is non-existent but well-commented example code is available and answers most of the questions. The examples folder in JMH code repository [9] is very informative.

B. Benchmarks

As benchmarks we used blocking request/reply (send) operations with varying size of payloads.

These benchmarks were used to find out approximations for latency and throughput for similar functionality written for Spring JMS and Akka.

We used blocking request/reply operations because this way the overhead of whole software stack was visible for entire round-trip.

For both platforms, the inherent asynchronous solution for "send" is to delegate the operation away from the running thread. In Spring JMS an available thread was used from the thread-pool or a new thread was launched if none was available. Akka's asynchronous "send" solution is to let the sending process be handled by the inner actors using general thread-pool in the kernel. In that sense the implementation details of Akka and Spring JMS do not differ.

1) Latency: The goal of latency benchmark was to find out how fast a message from one node reaches the second node. Benchmarks generated dummy loads of various sizes and the time of the "send" operation was measured per specific payload size.

2) Throughput: the goal of throughput benchmark was to measure how many operations per thread it is possible to perform per second. Benchmarks generated dummy loads of various sizes and the number successful "send" operations was counted.

IV. Test system architecture

Test system architecture is same for both latency and throughput measurements. In general the test system consists of two endpoints. Measurements are performed on a single endpoint - the one that initiates activity.

In figure 1 we show the system layout for Akka. The vertical line denotes the area where measurements are performed.

The Spring JMS solution contains one additional component - Message Broker. In figure 2 the Message Broker is

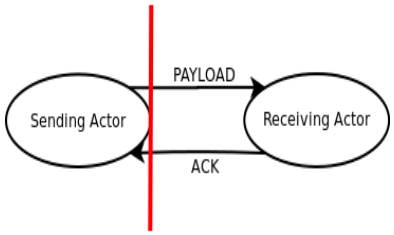

Fig. 1. Test system architecture for Akka.

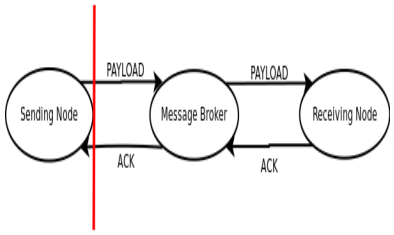

Fig. 2. Test system architecture for Spring JMS.

also shown. Again, the vertical line denotes the area where measurements are performed.

V. Implementation

A. Common functionality

Due to both platforms being so different in their nature, the common parts of the benchmarks were basically only ideological, such as run tests for packet sizes 1,10,100,...,1000000 of bytes. Uniform output was produced using JMH[9] framework.

JMH framework provides Java annotations for doing general initialization and in-line benchmark parametrization. These include measurement type (latency, throughput, sampling,..), time units, parallelization, fork counts and numerous other parameters.

A limitation is that if a benchmark with only slightly modified parameters is required, then that has to be implemented as a separate benchmarkable function and annotated accordingly.

JVM and code warmup is also provided by JMH. For that, the JMH enables to call a set of benchmarks before starting any measurements so that JIT, code in-lining in JVM and other optimizations. This is useful if real measurements are required for running application under heavy load.

B. Akka specifics

Java language was used for writing Akka-related code. For creating a close to reality situtation, a reply server was implemented using Akka - ”Receiving Actor” in 1 . That has launched before running any benchmarks.

Benchmarks are written using JMH framework that initializes and then calls simple Akka client implementation.

C. Spring JMS specifics

As for Akka, the code for Spring JMS benchmarks was written using Java and JMH.

For running Spring JMS-related benchmarks ActiveMQ[7] needs to be running also, see figure 2 for general overview. For the sake of simplicity a stable version of ActiveMQ was used with default parameters.

D. Parametrization

JVM and benchmarking parameters:

- Java 6
- Benchmarks are run in JVM with 2GB of heap space per endpoint.
- Each benchmark is run 10 for iterations.
- Each iteration lasts for 1 seconds.
- ActiveMQ was given 1GB of RAM.

E. System modification

During the benchmarking code development, the network settings on the machine used, needed to be modified. That was because of how the Linux kernel re-uses the sockets in the WAIT state. The re-use of sockets in WAIT state needed to be enabled and also the respective time-out value needed to be reduced[3].

F. Problems

Both Akka and Spring JMS needed some debugging.

Also JMH benchmarking framework gave some trouble.

1) Problems with Akka: one of the most tedious issue was the fact that Akka remoting capabilities erradically worked. That behaviour was due to the nature of how Akka and testing machine were configured. Due to the Akka still being actively developed, the default parameters change drastically between different versions of Akka. In Akka-s remoting library a timeout of 10 seconds is set for initializing remoting by default. That is done by reserving a TCP socket for remote incoming connections. However, the Linux system used (Ubuntu 12.04) had default delay of 60 seconds for socket reuse timeout.

2) Problems with Spring JMS: As with Akka, Spring JMS also needed reconfiguring the networking parameters on the system, but the modifications where all the same as for Akka.

3) Problems with JMH: JMH requires the benchmarks to be annotated with configuration parameters in the lowest subclass – e.g. it is not possible to write an abstract benchmarking class and then implement a subclass with minor modifications. This caused the duplication of benchmarking code.

VI. Results & Discussion

We performed measurements for blocking request/reply case. In the following we are reporting the results of measurements performed on Akka and Spring JMS.

[3] net.ipv4.tcp_tw_reuse = 1 # true
net.ipv4.tcp_fin_timeout = 10 # seconds

For Akka we built a single test system using asynchronous internal queries, for Spring JMS both synchronous and asynchronous internal queries. Initially we only developed single test-system for Spring JMS but it turned out that implementation using JmsTemplate was under-performing. More details follow below.

As we can see on the synchronous latency figure 3, the Akka outperforms JMS implementation by factor of 15-20. At first we thought that this difference can be attributed to the lack of message broker in the middle. In case of the benchmarking send, the heaviest operations are done for Akka in the client side when new data is pepared to be sent - a byte array of length x is reserved and put into wrapper class.

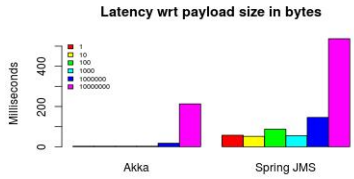

Fig. 3. Latency with respect to the payload size in bytes using JmsTempate.

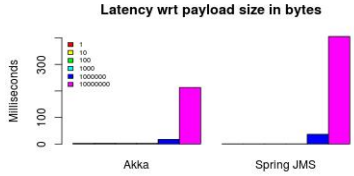

Fig. 4. Latency with respect to the payload size in bytes using custom request/reply implementation.

When looking at the throughput figure 5, we can see that Akka outperforms Spring JMS by factor of 10-15. However the strange shape of Spring JMS part of the plot indicates that there is something wrong - throughput is capped at arount 20 operations per second and for larger payload sizes that decreases even more.

This called for more investigation and we looked more deeply our solution.

The default class that we used for Spring JMS communication, JmsTemplate, is written in such a way, that by default it does send/receive operations very inefficiently. Namely, for each reply it creates a temporary Queue that the producer will listen on until the consumer is done processing the message. The new temporary queue creation operation is very expensive due to the fact that in addition to consumer and sender the Message Broker needs to allocate resources for the new queue.

To bypass this bottleneck we developed another implementation of request/reply functionality that reuses initially created temporary Queue for further replies as well.

On the figure 6 it is visible that Spring JMS is actually capable to outperform Akka. Only when payload sizes exceed certain limits then Akka performs better. Currently it is fully unclear where this difference comes from.

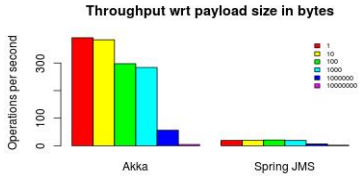

Fig. 5. Number of operations per second with respect to the payload size in bytes using JmsTempate.

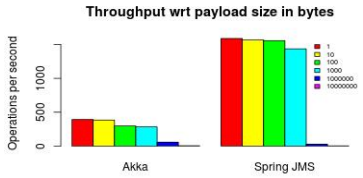

Fig. 6. Number of operations per second with respect to the payload size in bytes using custom request/reply implementation.

VII. Future work

Next we have to implement more specific use-cases and parts of real subsystem architecture for communication. That would give indication of how, and if at all, easily applicable given communication model is usable for reactive processing of data.

VIII. Conclusion

This work considered the differences of Akka and Spring JMS platforms. Some initial benchmarks were implemented to get more familiar with these platforms such as latency and throughput. We found out that Akka outperforms Spring JMS for certain payload sizes, but overall Spring JMS is faster.

References

[1] "The reactive manifesto." [Online]. Available: http://www.reactivemanifesto.org/

[2] "Spring jms." [Online]. Available: http://spring.io

[3] *Java Documentation Akka Documentation*. [Online]. Available: http://doc.akka.io/docs/akka/2.2.3/java.html

[4] "Erlang programming language." [Online]. Available: http://www.erlang.org/doc.html

[5] "DSpace@MIT: ACTORS: a model of concurrent computation in distributed systems." [Online]. Available: http://dspace.mit.edu/handle/1721.1/6952

[6] J. Armstrong, "A history of erlang," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 6–1–6–26. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238850

[7] "Apache ActiveMQ – index." [Online]. Available: http://activemq.apache.org/

[8] "caliper - microbenchmarking framework for java." [Online]. Available: https://code.google.com/p/caliper/

[9] "OpenJDK: jmh." [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/

[10] "Maven - welcome to apache maven." [Online]. Available: http://maven.apache.org/

Fig. 7. Sample configuration snippet for deploying Akka actor on remote system.

```
akka.actor.deployment {
  /remoteActor {
    remote = "akka.tcp://RemoteMachine@10.0.0.42:2552"
  }
}
```