Yet Another Akka Benchmark

August 10, 2010 - Patrik Nordwall (https://blog.jayway.com/author/patriknordwall/) -
Architecture (https://blog.jayway.com/category/architecture/) / Java (https://blog.jayway.com/category/java/) -
5 Comments (https://blog.jayway.com/2010/08/10/yet-another-akka-benchmark/#comments)

I have exercised my Scala (http://www.scala-lang.org/) and Akka (http://akkasource.org/) skills by creating a sample application in the trading domain. I have experience from developing trading systems so even though the sample is simplified it is rooted in real world architecture.
I have found it interesting to compare performance of different technical solutions with this sample application. Therefore, several concrete implementations of the trading system are implemented and benchmarked against each other.

1. Ordinary synchronous method invocations.
2. Scala Actors
3. Akka Actors

I can tell you right away that the performance of Akka Actors is outstanding compared to Scala Actors.

Before looking at the benchmark I will briefly describe the sample application.

A trading system is essentially about matching buy and sell orders. A limit order is an order to buy a security at not more, or sell at not less, than a specific price. For example, if an investor wants to buy a stock, but doesn't want to pay more than $20 for it, the investor can place a limit order to buy the stock at $20 "or better". There are many other types of orders and special constraints. The sample is only handling plain limit orders.

Orders that are away from the current best price in the market are collected in an order book for the security, for later execution.

A matching engine manages one or more order books, i.e. the marketplace is sharded by order book. The matching engines holds current state of the order books. Clients connect to an order receiver service, which is responsible for routing the order to the correct matching engine. The order receiver is stateless, and the clients can use any order receiver independent of order book.

For redundancy, the matching engines work in pairs. Each order is processed by both matching engines. The order is also stored in a persistent transaction log, by both matching engines. In a real setup the primary and standby matching engines are typically deployed in separated data centers.

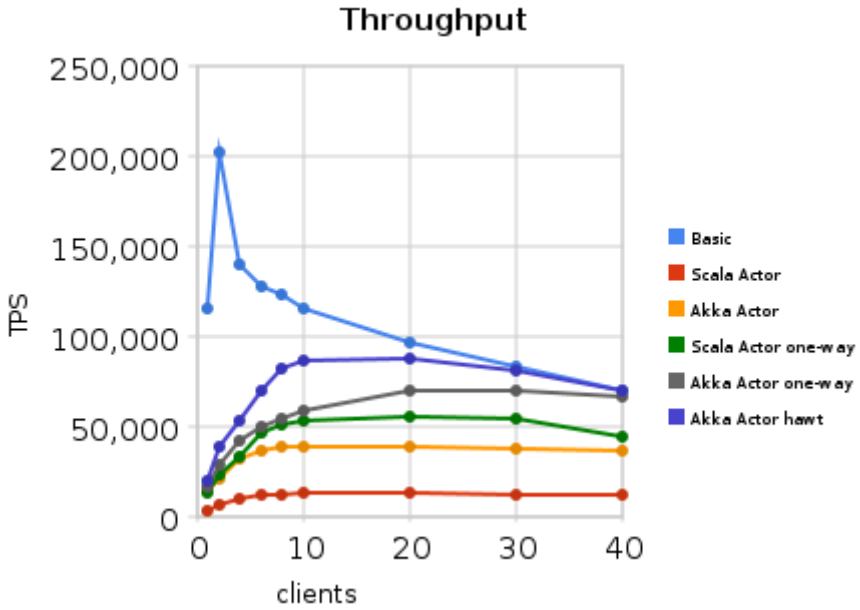
Now, over to the benchmark. The test scenario put buy and sell orders in 15 order books, divided in 3 matching engines. The orders are at different price levels, so an order book depth is built up, but in the end all orders are traded and that is verified by the JUnit test running the benchmark.

The scenario was run at different load, by varying the number of simulated clients from 1 to 40.

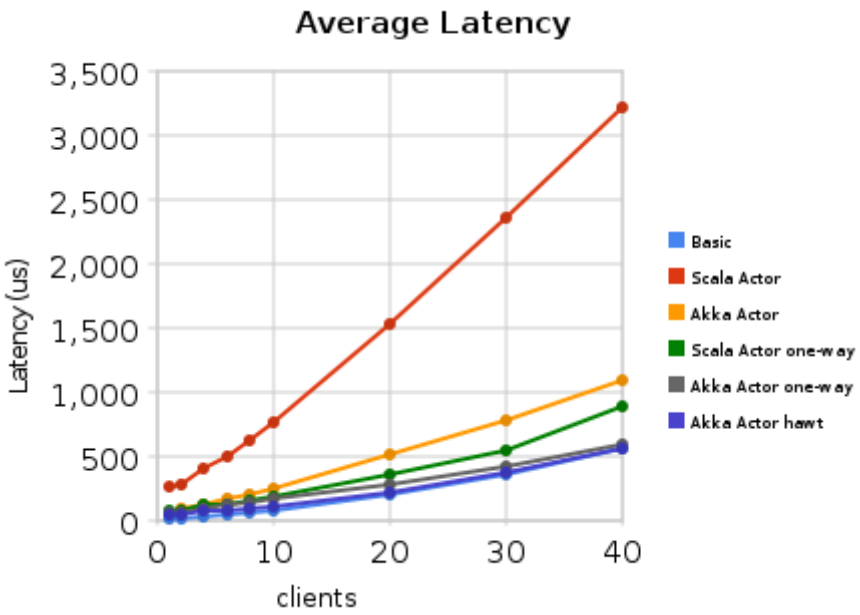
The benchmark results illustrated here were performed on a real 8 core box (dual cpu Xeon 5500 machine, 2.26 Ghz per core).

(https://blog.jayway.com/category/aspect-oriented-programming/)
Assistants
(https://blog.jayway.com/category/assistants/)
Augmented Reality
(https://blog.jayway.com/category/augmented-reality/)
Automotive
(https://blog.jayway.com/category/automotive/)
C++
(https://blog.jayway.com/category/c/)
Cloud
(https://blog.jayway.com/category/cloud/)
Cocoa
(https://blog.jayway.com/category/cocoa/)
Competence Development
(https://blog.jayway.com/category/competence-development/)
Data Science
(https://blog.jayway.com/category/data-science/)
Design
(https://blog.jayway.com/category/design/)
DevOps
(https://blog.jayway.com/category/devops/)
Dynamic languages
(https://blog.jayway.com/c

Here are the result of processing 750000 orders at each load level.



(http://blog.jayway.com/wp-content/uploads/2010/08/throughput3.png)



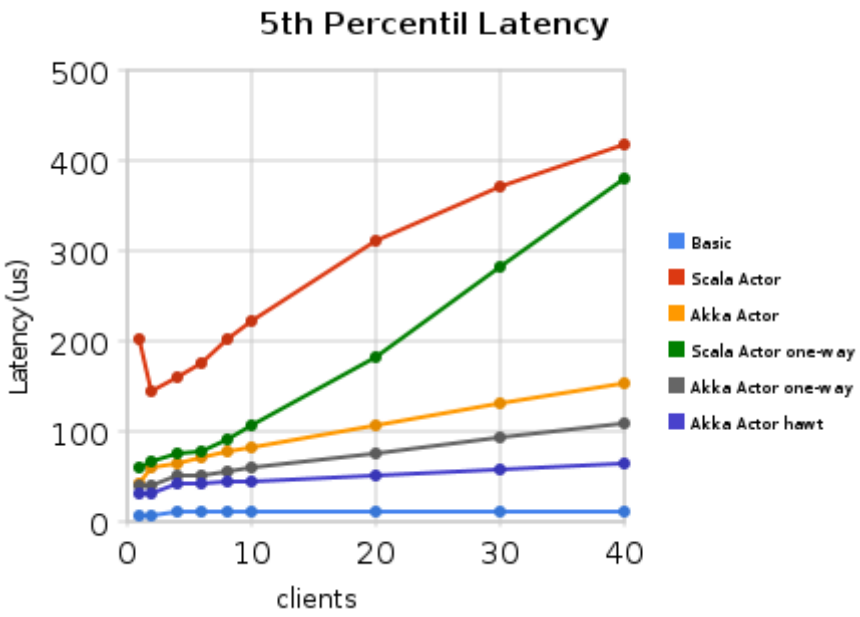
(http://blog.jayway.com/wp-content/uploads/2010/08/average_latency3.png)

The Basic solution uses ordinary synchronous method invocations. It is extremely fast, but not an option for a true scalable solution. Asynchronous message passing is a better alternative for scaling out on multi-core or multiple nodes.

ategory/dynamic-languages/)
Embedded (https://blog.jayway.com/category/embedded/)
Events (https://blog.jayway.com/category/events/)
Functional programming (https://blog.jayway.com/category/functional-programming/)
Generics (https://blog.jayway.com/category/swift/generics/)
Graphics (https://blog.jayway.com/category/graphics/)
iOS (https://blog.jayway.com/category/ios/)
IoT (https://blog.jayway.com/category/iot/)
Java (https://blog.jayway.com/category/java/)
JavaScript (https://blog.jayway.com/category/javascript/)
Kotlin (https://blog.jayway.com/category/kotlin/)
Linux (https://blog.jayway.com/category/linux/)
Other (https://blog.jayway.com/c

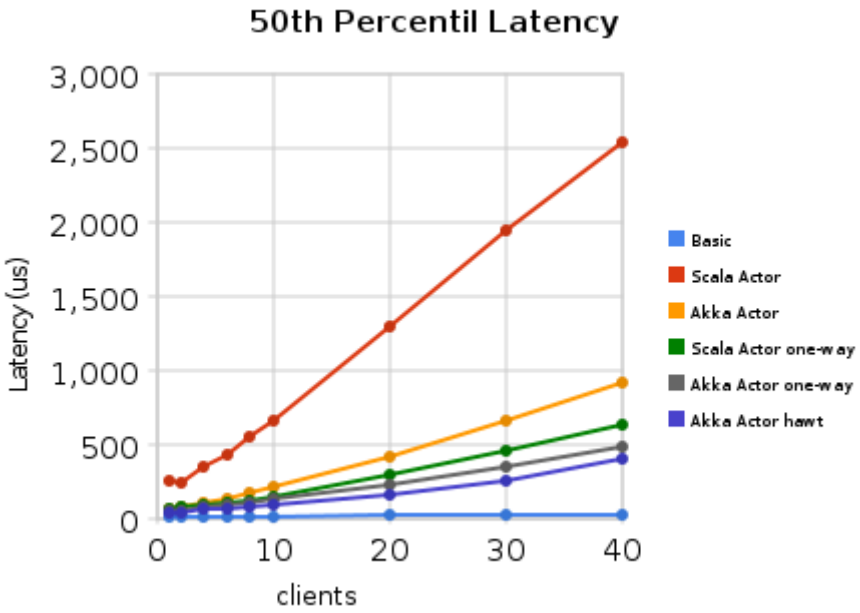
In the Scala and Akka Actors solutions the clients send each order message to an order receiver and waits on the response Future (!? operator in Scala and !! in Akka). The order receiver forwards the request to the matching engine responsible for the order book, i.e. the order receiver thread/dispatcher can immediately be used for next request. The matching engine sends the order message to the standby and both matching engines process the matching logic and transaction logging in parallel. Acknowledgment is replied to client when both are done.

The benchmark results shows that Akka Actors are able to process three times as many orders compared to Scala Actors at the the same load. Similar result with latency. The latency of Akka Actors is one third of Scala Actors. This holds for low load also. Average latency is not always the best measure, so let us look at some percentiles.

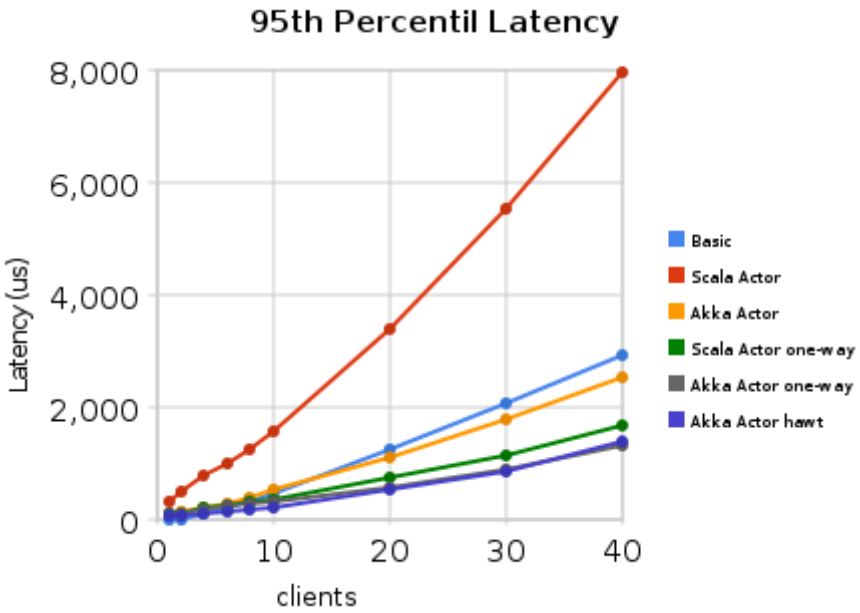


(http://blog.jayway.com/wp-content/uploads/2010/08/5th_percentil_latency3.png)

ategory/other/)
python (https://blog.jayway.com/category/python/)
React (https://blog.jayway.com/category/web/react/)
Scala (https://blog.jayway.com/category/scala/)
Security (https://blog.jayway.com/category/security-2/)
Swift (https://blog.jayway.com/category/swift/)
SwiftUI (https://blog.jayway.com/category/swift/swiftui/)
Testing (https://blog.jayway.com/category/testing/)
Tips & Tricks (https://blog.jayway.com/category/tips-and-tricks/)
Tools & Workflows (https://blog.jayway.com/category/tools-and-workflows/)
Tutorial (https://blog.jayway.com/category/tutorial/)
Uncategorized (https://blog.jayway.com/category/uncategorized/)
Unity (https://blog.jayway.com/category/unity/)



(http://blog.jayway.com/wp-content/uploads/2010/08/50th_percentil_latency3.png)



(http://blog.jayway.com/wp-content/uploads/2010/08/95th_percentil_latency3.png)

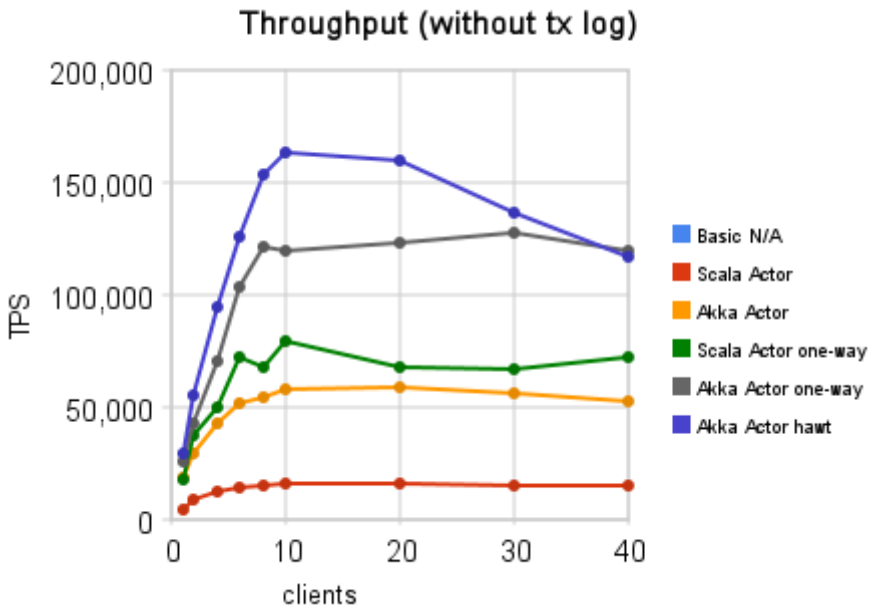
Operations that are waiting for a Future to complete has been used when sending messages. This has a scalability price tag, since the thread is blocked while waiting for the Future to complete. Better scalability can be achieved with one-way message passing, which is illustrated by the Scala/Akka Actor one-way solutions. It uses bang operation (!) for sending of all messages.

User Experience (https://blog.jayway.com/category/user-experience/)
VR/AR (https://blog.jayway.com/category/vrar/)
Wearables (https://blog.jayway.com/category/wearables/)
Web (https://blog.jayway.com/category/web/)

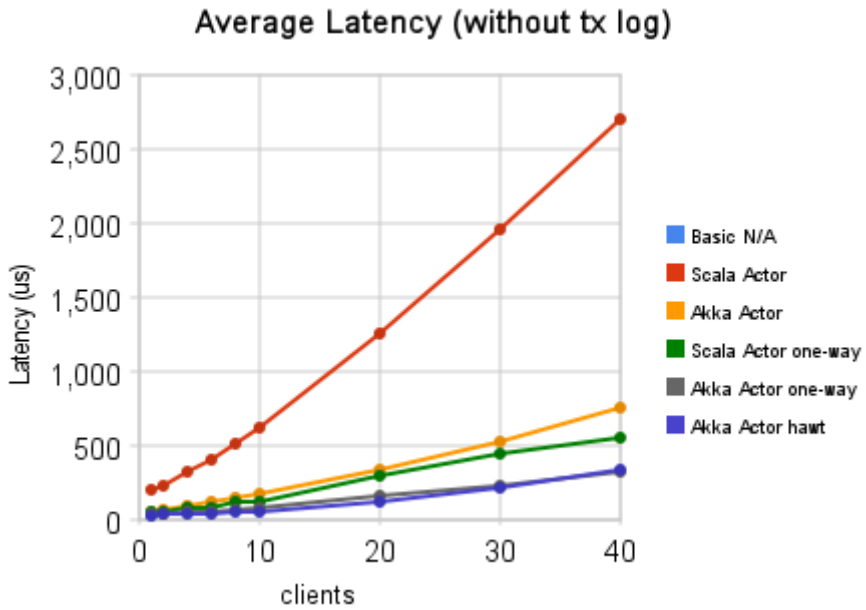
TAGS

.Net (https://blog.jayway.com/tag/net/) Android (https://blog.jayway.com/tag/android/) aop (https://blog.jayway.com/tag/aop/) automated testing (https://blog.jayway.com/tag/automated-testing/) aws (https://blog.jayway.com/tag/aws/) azure (https://blog.jayway.com/tag/azure/) C# (https://blog.jayway.com/t

The matching engines writes each order to a transaction log file. This is a blocking IO bottleneck. To push the test of message passing one step further the benchmark has also been run without the transaction log. The Akka solution shines even more. More than three times higher transaction rate compared to Scala Actors at the the same load, when using solution based on sending messages and waiting for reply. For the one-way message passing solutions the Akka Actors are two times faster than Scala Actors.



(http://blog.jayway.com/wp-content/uploads/2010/08/throughput_without_tx_log3.png)



(http://blog.jayway.com/wp-content/uploads/2010/08/average_latency_without_tx_log3.png)

ag/c/) clojure (https://blog.jayway.com/tag/clojure/) conference (https://blog.jayway.com/tag/conference/) frameworks (https://blog.jayway.com/tag/frameworks/) functional programming (https://blog.jayway.com/tag/functional-programming/) git (https://blog.jayway.com/tag/git/) http (https://blog.jayway.com/tag/http/) iOS (https://blog.jayway.com/tag/ios/) iphone (https://blog.jayway.com/tag/iphone/) Java (https://blog.jayway.com/tag/java/) javascript (https://blog.jayway.com/tag/javascript/) jayview (https://blog.jayway.com/tag/jayview/) junit (https://blog.jayway.com/tag/junit/) maven (https://blog.jayway.com/tag/maven/) metro

Akka has great flexibility when it comes to specification of different dispatching mechanisms. The Akka Actor hawt is included in the benchmark as a comparison with the Akka Actor one-way solution. It uses the HawtDispatch threading library (http://hawtdispatch.fusesource.org/) which is a Java clone of libdispatch. The last test without transaction log shows that HawtDispatcher has slightly better performance than the event-based dispatcher that has been used for Akka Actor one-way.

The complete source code for the sample application is located at: http://github.com/patriknw/akka-sample-trading (http://github.com/patriknw/akka-sample-trading)

To run the benchmark yourself you can download (http://github.com/patriknw/akka-sample-trading/downloads) and unzip the distribution, containing all needed jar files. Included README (http://github.com/patriknw/akka-sample-trading) describes how to run the tests.

Update note Aug 15: Added Scala Actor one-way solution and new description of how to run the benchmark.

Update note Aug 22: New benchmark run on real 8 core box.

› THIS POST HAS 5 COMMENTS

Pingback: Tweets that mention Yet Another Akka Benchmark — Jayway Team Blog -- Topsy.com (http://topsy.com/www.jayway.com/2010/08/10/yet-another-akka-benchmark/?utm_source=pingback&utm_campaign=L2)

Erik Engbrecht (http://erikengbrecht.blogspot.com) 16 AUG 2010 REPLY