# Comparing ActiveMQ and Akka

## A practical comparison by implementing the Leibniz Formula

*Markus Hösel, BSc.*
TDDD05 Component Based Systems
Linköping University
Linköping, Sweden
Marho348@ad.liu.se

***Abstract*— This paper gives a comparison of ActiveMQ and Akka. Both toolkits are event message based and can in practice be used to loosely couple components in a software system, but their fundamental concepts are quite different. A theoretical description of the concepts is therefor given and by implementing Leibniz's Formula for calculating Pi differences and similarities in code will be highlighted. The implementation will be tested under different parameter settings and the results will be outlined. A discussion will be given and one will see that Akka handles the message processing more effective than ActiveMQ and that both toolkits are able to increase the performance significant by following a concurrent approach.**

***Keywords—ActiveMQ; JMS; Akka, MOM, message oriented middleware, Actor;***

## I. INTRODUCTION

The goal of this paper is to give a comparison between ActiveMQ and Akka. ActiveMQ is a message-oriented middleware which is compliant with the Java Message Service [2]. Akka makes use of the Actor model, a mathematical model of concurrent computation [3]. Both concepts are quite different. Therefore the paper describes the concepts, shows an implementation and analyses the results by executing it. The structure of the paper is therefore as follow:

Chapter 2 introduces ActiveMQ and its fundamental concepts it is build upon. Chapter 3 introduces Akka and the Actor model. The actual implementation of both ActiveMQ and Akka to calculate Pi using Leibniz's formula is outlined in Chapter 4 and in Chapter 5 the results are shown and discussed. A conclusion is given in Chapter 6. The paper ends by referring to Future Work in Chapter 7.

## II. ACTIVEMQ CONCEPTS

ActiveMQ is a message-oriented middleware (MOM) which fully implements the Java message service (JMS) [2]. This chapter thus introduces the terms MOM, JMS and ActiveMQ.

### *A. MOM*

A MOM provides a clean method of communication between disparate software entities by implementing messaging capabilities. In such a system so called clients exchange data by sending and receiving messages from each other. In doing so the clients connect to a message brokers, which acts as an intermediary and is responsible for the routing of the messages [7].

A client can therefore be seen as a component, which is a self-contained piece of software interacting with the rest of the system through its interface. This approach allows to build loosely coupled, distributed systems in which a client can be easily changed or replaced without having to interfere with other parts of the system [2][5].

Communication in a MOM system is made asynchronously and is non-blocking. Therefore a client can immediately continue processing after it sent a message, without being required to block and wait for a response. It is the message brokers' job to store and deliver the messages to the receiver. This approach is especially beneficial when the receiver is busy or not available. Without a message broker the sender would have to block and wait until the receiver can process the message which could delay the whole system [4].

A MOM typically supports two different messaging models [17]:

- Point-to-Point: In this model the broker implements a first-in first-out message queue in which the messages are stored. Figure 1 shows the messaging model. Producer can send message to this queue which are then routed by the broker to the consumers. A particular message can only be consumed by one consumer. Having multiple consumers retrieving messages from the same queue allows efficient load balancing, which is the main advantage of this model.

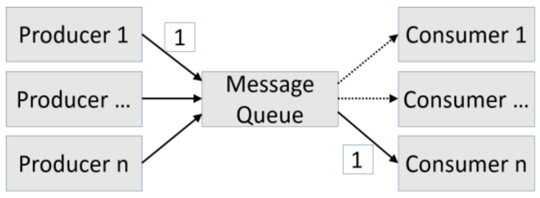

Figure 1: Point-to-Point model

- Publish/Subscribe: In this model clients publish and subscribe messages according to a specific topic. Figure 2 shows the Publish/Subscribe model where on the one side Publisher1 sends a particular message to a specific topic and on the other side all clients subscribed to the topic consume this particular message. In this model the message broker is responsible to route the messages to all clients on the basis to which topics they subscribed. This model mostly extends basic functionality by providing message filtering or subscription management.

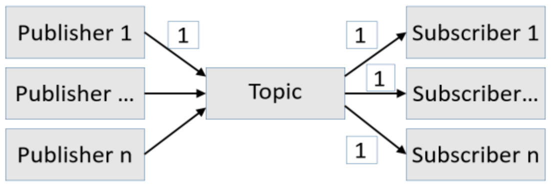

Figure 2: Publish/Subscribe model

### B. JSM

The Java Message Service is a MOM API for creating, sending, receiving and reading messages produced and consumed by clients [1]. It is written in Java and part of the Java Enterprise Edition. The specification and its API are standardized in JSR 914 and define a common set of interfaces and associated semantics that allow programs written in Java programming language to communicate with multiple heterogeneous messaging systems [8]. In order to use JMS a JMS provider is needed, which manages the topics, queues and sessions accordingly [18]. Popular providers are ActiveMQ, FuseMQ or ZeroMQ.

### C. ActiveMQ

ActiveMQ is an Open Source message-oriented middleware hosted by the Apache Software Foundation. It fully implements the JMS Specification [1] and adds additional features on top of that. Such features are high availability, performance, scalability, reliability and security for enterprise messaging [4], and the support for various network protocols such as HTTP, HTTPS, SSL, TCP, UDP, STOMP, XMPP [2].

The ActiveMQ message broker itself is written in Java, but ActiveMQ provides cross language clients for the most common languages like C, C++, Erlang, Go, Haskell, NodeJS, Perl, Python, Tcl/Tk and has also JMS-, REST- and WebSocket Interfaces to connect to [2]. ActiveMQ has further full support for pluggable persistence through JDBC but has also its own persistence layer called KahaDB [6].

## III. Akka Concepts

### A. Actor Model

The actor model was developed in order to solve computing problems following a concurrent and distributed approach. In such a system multiple actors interact with each other through messages [3]. Similar to the clients in a MOM an actor acts as a component – a self-contained piece of software with an interface. But actors are more powerful. In response to a received message an actor has the following possibilities [9]:

- Message Handling: Respond or forward the message or send new messages to other actors.
- Supervision: Create new actors and manage them.
- State Machine: Change its internal behavior

Figure 3 shows that the actors interact through asynchronous messaging. Although every actor has a mailbox which adds an additional layer, it can be said that the communication is done directly between the actors [9].

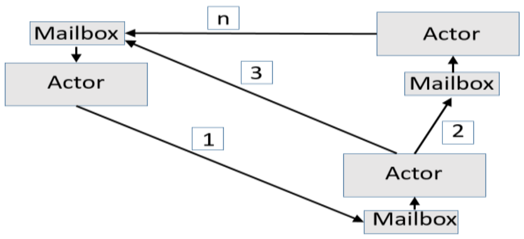

Figure 3: The Akka actor model

Actors implement a “share nothing” approach, which means that they do not share their mutable states, e.g. variables. Based on that, actors do not have the need to lock their state, one very important aspect in reference to performance and throughput.

Actors also implement a State Machine which allows them to transition from one internal state to another in response to a message, and by doing so changing its internal behavior for future messages.

Another great feature of actors is, that they can create other actors, and are responsible for managing them. This means that they take over supervision. They can delegate tasks to their subordinate actors and are responsible for handling failures of them. Subordinate actors can be restarted, stopped, resumed or destroyed by their supervisor.

### B. Akka

Akka is a toolkit which allows to create distributed, concurrent, fault-tolerant and scalable applications by using actors [3]. Furthermore the applications are scalable, resilient and responsive in means of the Reactive Manifesto [10]. Akka is Open Source and hosted by Lightbend (formerly Typesafe) and runs on the Java Virtual Machine.

It is written in Scala and has language bindings for Java and Scala. Its main focus is on Actors, Fault Tolerance, Location Transparency and Persistence. The actors are very lightweight event-driven processes, therefore one can instantiate several million actors per GB of heap memory [3].

In order to scale accordingly it's also possible for an actor to create new sub actors and destroy them. The Supervisor hierarchies implement a "let-it-crash" semantics where sub actors can be stopped by either calling a postStop hook to destroy an actor after the current message is processed or sending a message of type PoisonPill, which will be inserted in the actors' mailbox and gets processed according to the FIFO principle [9].

If a supervisor can't decide how to react to a failure of a child actor it can fail itself, which delegates the supervision one layer higher. It is this approach of failing, restarting and delegating why it is called "let-it-crash" semantic [19].

## IV. Implementation

This chapter describes the mathematical formula used for testing, the environment in which the program is tested, the structure of the program itself and the actual implementation in Java for ActiveMQ and Akka. Moreover the main differences and similarities are outlined.

### A. Leibniz Formula

In order to test both toolkits in respect to efficiency of message handling and concurrency a program is written which calculates the value of Pi. The formula used is called Leibniz formula [11] [12] and is given in (1).

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \ldots = \frac{\pi}{4} \qquad (1)$$

By using this formula the calculation can be split into several chunks of discretionary size, with each chunk first getting calculated independently and then being accumulated to the total result.

For example, one could calculate up to the 6th element by using two chunks of each three elements and accumulate both to the total number. The necessary steps are:

- 1st chunk: ( 1 - 1/3 + 1/5 ) * 4 = 3.4667
- 2nd chunk: ( -1/7 + 1/9 – 1/11 ) * 4 = - 0.4906
- Total = 3.4667 – 0.4906 = 2.9761

Because each chunk gets calculated individually, they can be calculated in parallel. The result of each chunk can then be send via a message to a process which accumulates the total.

### B. Environment

The program is executed in a virtual environment existing of one virtual machine with two processors and 4 GB RAM. Ubuntu 15.10 is used as operating system. The ActiveMQ implementation and the Akka implementation are written in Java using OpenJDK 1.8.0 Update 66. Apache ActiveMQ 5.8.0 and Akka 2.0.2 are used for the two individual parts. The source code is published on Github[1].

1 https://github.com/hoeselm/CBS_ActiveMq_vs_Akka

### C. Program Structure

While the actual implementations of ActiveMQ and Akka differs, the core structure is the same. This paragraph lists the components used to process the data and the messages used for the interprocess communication:

- Worker Message: Plain Old Java Object (POJO) with two member variables called *start_value* and *number_of_elements*. The *start_value* variable describes the starting point for the calculation of a chunk and the *number_of_elements* variable defines the number of elements to process.
- Collector Message: POJO with one member variable called *sum*. This variable holds the result of an individual chunk.
- Creator: Component used to split up the calculation into chunks. It takes as a parameter the number of chunks and instantiates for each chunk a Worker Message. The *start_value* and the *number_of_elements* variable are initialized accordingly and the messages are sent to the system.
- Worker: Component used to process the chunks. Multiple instances are created in order to process the individual chunks in parallel. Each Worker is listening for Worker Messages. For every Worker Message retrieved a calculation based on the information in the message is performed and the result is saved in the member variable *sum* of a Collector Message. These Collector Messages are then sent back to the system.
- Collector: Component used to calculate a final result. It is therefore listening for Collector Messages. For each message received the *sum* value of that message is unpacked and added to the total number of Pi.

Figure 4 shows the message flow between the components.

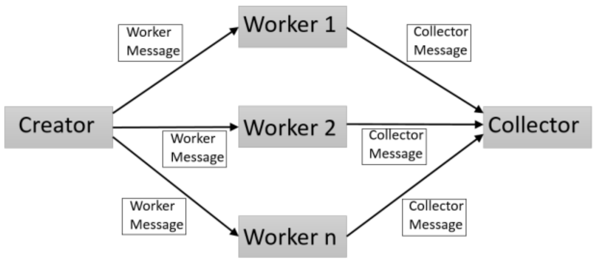

Figure 4: Message Flow between Components

### D. ActiveMQ Implementation

ActiveMQ follows the outlined program structure very closely. The only difference is that the *Worker* component and the *Collector* component refer to one additional component which actually listens for incoming messages. These components are called *Worker Listener* and *Collector Listener*. Consequently the following classes are implemented:

- Message Broker: Handles the queues and messages and is started via command line, thus is not part of the actual Java implementation.
- WorkerMessage.java: Implements the *Worker Message* as a POJO.
- CollectorMessage.java: Implements the *Collector Message* as a POJO.
- App.java: This is the main file. It first specifies the number of *Workers* to use, the number of *Worker Messages* to create and how many elements should be calculated per each message. It then creates the components, with each one running in a separate thread.
- Creator.java: Implements the *Creator* and sets up a connection to the message broker as producer, creates the *Worker Messages* and sends them to the broker.
- Worker.java: Implements the *Worker* and sets up a connection to the message broker as consumer and specifies an instance of *Worker Listener* as listener for incoming *Worker Messages*.
- WorkerListener.java: Implements the *Worker Listener* and sets up a connection to the message broker as producer, listens for incoming *Worker Messages*, performs the calculation and sends the result wrapped in a *Collector Message* to the message broker.
- Collector.java: Implements the *Collector* and sets up a connection to the message broker as consumer and specifies an instance of *Collector Listener* as listener for incoming *Collector Messages*.
- CollectorListener.java: Implements the *Collector Listener*, listens for incoming *Collector Messages* and adds the individual results of each *Collector Message* to the final number of Pi.

In order to process the incoming *Worker Messages* in parallel, the ActiveMQ implementation makes use of the Point-to-Point model [17].

The message sending and retrieving process works through a session instance. This is done by creating an instance of a *connectionFactory* first, and using it to create a *connection* instance. Through the *connection* instance the *session* can be created. This is shown in Example 1 and needs to be implemented for each component which should handle messages.

```
// create a ActiveMQConnection Factory instance
connectionFactory = new
ActiveMQConnectionFactory(messageBrokerUrl);
// create connection to the message broker
connection = connectionFactory.createConnection();
// create a session based on the connection, with
transaction handling false and auto acknowledged
session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
```

Example 1: Establish a session to the message broker

Depending on weather a component should send or retrieve messages, a producer or a consumer instance is created by using the existing session instance. This is shown in Example 2 and Example 3 by using the Work Queue which is shared between Creator and Worker. The same principle is used for the Collector Queue as well which is shared between Worker and Collector.

```
// messages are send to the Work queue
destination = session.createQueue("Queue.Work");
// create a message producer using the session
producer = session.createProducer(destination);
```

Example 2: Creating a producer

```
// queue definitions
destination = session.createQueue("Queue.Work");
// create a message consumer using the session
consumer = session.createConsumer(destination);
```

Example 3: Creating a consumer

Example 4 lays out how the Creator uses an instance of the producer defined in Example 2 to send a Worker Message to the message broker. It first creates a *Worker Message* which is then wrapped in an *Object Message*. This *Object Messages* can further be used to append metadata, in this example a message id with the value 1 is added. Finally the message is sent.

```
// create message
WorkerMessage worker_message = new
    WorkerMessage(start_value, number_of_el);
ObjectMessage object_message =
    session.createObjectMessage(worker_message);
object_message.setIntProperty("message_id", 1);
// send message
producer.send(object_message);
```

Example 4: Creating and sending a message

The job of the *Worker* is to register the *Worker Listener*. This is done in Example 5.

```
// create a listener
consumer.setMessageListener(new
WorkerListener());
```

Example 5: Add a listener

The actual consumer for the *Worker Messages* is implemented in the *Worker Listener* by overriding a method called *onMessage*, which takes a message as parameter.

```
public void onMessage(Message message) {

// parse worker message and calculate result
ObjectMessage object_message = (ObjectMessage)
message;
WorkerMessage worker_message = (WorkerMessage)
object_message.getObject();

// calculate chunk
double sum = calculate_pie(worker_message);

// create sum message
 CollectorMessage collector_message = new
CollectorMessage(sum);
ObjectMessage collector_object_message =
session.createObjectMessage(collector_message);
collector_object_message.setIntProperty("messag
e_id", message_id);

// send sum message
producer.send(collector_object_message);

}
```

Example 6: onMessage method of worker listener instance

This message is first casted to an *Object Message* and then unwrapped to get the *Worker Message*. The *Worker Message* is used to perform the calculation and a *Collector Message* is created with the result. The *Collector Message* is then send to the message broker. The code for this operations is outlined in Example 6.

The *Collector* registers the *Collector Listener* the same way the *Worker* did for the *Worker Listener* in Example 5.

Example 7 outlines the *onMessage* method for the *Collector Listener*, which shows the final step performed to calculate Pi.

```
public void onMessage(Message message) {

// parse worker message and calculate result
ObjectMessage object_message = (ObjectMessage)
message;
CollectorMessage sum_message =
(CollectorMessage) object_message.getObject();

// calculation
pi += sum_message.getSum();
}
```

Example 7: onMessage method of collector listener instance

An instance of the message broker itself is started on the command line and is then listening on Port 61616 for incoming messages. Example 8 shows the command to start the message broker in a Linux environment as well as a command which outputs network statistics.

```
user@host:~/apache-activemq-5.8.0$ ./bin/activemq start
user@host:~/apache-activemq-5.8.0$ netstat –tpl
Proto  Recv-Q  Local Address   Foreign Address   State       PID
Tcp6   0       ::61616         :::*              LISTEN   2129/java
```

Example 8: Starting a message broker

## *E. Akka Implementation*

Akka follows exactly the outlined program structure, but an additional start message is used to trigger the first component. The following classes exist in the Akka implementation:

- StarterMessage.java: Implements a *Start Message* as POJO with no member variable at all.
- WorkerMessage.java: Implements the *Worker Message* as a POJO.
- CollectorMessage.java: Implements the *Collector Message* as a POJO.
- App.java: Main file, which first specifies the number of workers to use, the number of messages to create and the number of elements to calculate per message. It then creates the actor system itself as well as the components as actor instances and starts the program by creating a *Creator* instance and sending it a *Starter Message.*
- Creator.java: Creates the *Worker* instance and the *Collector* instance. When receiving a *Starter Message* it also creates the *Worker Messages* and sends them to the worker.
- Worker.java: Listens for *Worker Messages*, performs the calculation for each individual received message and sends the result wrapped in a *Collector Message* to the *Collector*.
- Collector.java: Listens for *Collector Messages* and adds the individual result of each message to the final number of Pi.

In order to process the messages an actor system is set up in the main file. This can be seen as the equivalent to the message broker in the ActiveMQ example and is outlined in Example 9.

```
// Create an Akka system
ActorSystem system =
ActorSystem.create("PiCalculation");
```

Example 9: Starting an Akka system

In the main file is furthermore the Creator actor instantiated and the Start Message sent. This process is outlined in Example 10.

```
// Create a creator actor
ActorRef creator = system.actorOf(new Props(new
UntypedActorFactory() {
    public UntypedActor create() {
        return new Creator(
          start_time,
          actor_count,
          message_count,
          number_of_elements_per_message);
    }
}), "creator");

// Start calculation
StartMessage start_message = new StartMessage();
creator.tell(start_message);
```

Example 10: Creating an actor and send a message

The Creator actor is instantiated by using a so called Props class, which is a configuration class and allows to specify different options. Here we pass an instance of an UntypedActorFactory and a name to the Props object. The UntypedActorFactory itself acts as a wrapper to the Creator class.

After this definition a Start Message is created and sent to the Creator actor by using the tell() method on the object.

The Creator actor then instantiates a Collector actor and several Worker actors. For the Worker actors a RoundRobinRouter is used, which allows to distribute the Worker Messages in a round robin fashion to the Worker actors. Example 11 outlines the code for the instantiation of the Worker actors. The instantiation of the Collector actor follows the same principle as outlined in Example 10 and is therefore not shown.

```
// create the worker using round robin router
workers = this.getContext().actorOf(new
Props(new UntypedActorFactory() {
    public UntypedActor create() {
        return new Worker(collector);
    }
}).withRouter(new
RoundRobinRouter(actor count)), "workers");
```

Example 11: Creating a RoundRobinRouter

Lastly the Creator actor implements a onReceive() method. As soon as the Creator receives an instance of the Start Message it creates the Worker Messages and sends them to the Worker actors by using the RoundRobinRouter defined in Example 11.

The Worker actor implements also a onReceive() method, and performs a calculation for each Worker Message received. The result is than wrapped in a Collector Message and sent to the Collector. These steps are shown in Example 12.

```
// message listener
public void onReceive(Object message) {

    // listen for worker messages
    if (message instanceof WorkerMessage) {

        // parse message
        WorkerMessage worker_message =
(WorkerMessage) message;
        // calculate result
        double sum =
calculate_pie(worker_message);
        // create sum message
        CollectorMessage sum_message = new
CollectorMessage(sum);
        // send sum message
        collector.tell(sum_message, getSelf());
    }
}
```

Example 12: onReceive method of worker actor

Finally the Collector actor listens for Worker Messages in its onReceive() method and adds each individual result to the total number of Pi as shown in Example 13.

```
// message listener
public void onReceive(Object message) {
    // listen for worker messages
    if (message instanceof CollectorMessage) {
        // parse message and calculate result
        CollectorMessage collector_message =
(CollectorMessage) message;
        // calculation
        pi += collector_message.getSum();
    }
}
```

Example 13: onReceive method of collector actor

## F. *Differences and simularities*

One major difference in the implementation is that for ActiveMQ one writes independent components, with each component running as separate Java program or, as implemented in this example, as one Java program which makes use of multiple threats. Furthermore each component has to create either producer or consumer object, and therefore must also create a connectionFactory, a connection and a session object. Before instantiating the components, a message broker has to be started. This can be done from a command line.

In contrast to that an Akka program is implemented as one solution. It's the job of the toolkit to create the actors in a concurrent non-blocking way, rather than the programmer's task to handle multiple threats or multiple Java programs. There is also no need to start a message broker by yourself, as the message handling process is created within the Akka system.

Aside from these differences the implementations are very similar. Both implementation can use POJOs to wrap necessary data in a message.

For the actual message sending process both implementation define a method on an object which represents the destination. This method is named *send()* in the ActiveMQ implementation and *tell()* in the Akka implementation, with both methods taking the message to deliver as their argument.

Moreover, both toolkits implement a method called *onMessage()* in the ActiveMQ implementation and *onReceive()* in the Akka implementation in order to receive messages. Both methods take a message object as parameter. While ActiveMQ already defines the messages to retrieve on the connection object, Akka has to use an if-else statement to check for the Type of the message in the method itself.

## V. Measurements

The ActiveMQ and Akka implementation are executed multiple times with different configuration settings for each execution. The measurement spans the whole process from creating the components till the last message is received by the Collector. Each run increases either the number of messages, workers or elements processed per message. The results for ActiveMQ are shown in Table 1 and 2, for Akka in Table 3 and 4.

Table 1 displays the results of the ActiveMQ implementation for 1, 10 and 100 active worker instances processing 100, 1000, 10 000, 100 000 messages with each message triggering a calculation of Pi for 10 elements. Table 2 shows the result for one million elements processed elements per message. The tables are shown in Results and a discussion of the results is made in Discussion. Table 3 shows the results of the Akka implementation by using the same parameters as for Table 1. Table 4 displays the results for Akka by using the same parameters as for Table 2.

### A. *Results*

TABLE I. ACTIVEMQ WITH 10 ELEMENTS PER MESSAGE

| **10 elements per message** | **Active Workers** | | |
|---|---|---|---|
| **Messages processed** | ***1*** | ***10*** | ***100*** |
| 100 | 1.69 sec. | 1.42 sec. | 1.63 sec. |
| 1000 | 5.53 sec. | 4.81 sec. | 5.38 sec. |
| 10 000 | 23.92 sec. | 26.42 sec. | 23.33 sec. |
| 100 000 | 237.59 sec. | 135.51 sec. | 137.65 sec. |

TABLE II. ACTIVEMQ WITH 1 MILLION ELEMENTS PER MESSAGE

| **1 million el. per message** | **Active Workers** | | |
|---|---|---|---|
| **Messages processed** | ***1*** | ***10*** | ***100*** |
| 100 | 2.30 sec. | 1.59 sec. | 1.62 sec. |
| 1000 | 12.19 sec. | 6.66 sec. | 8.65 sec. |
| 10 000 | 101.86 sec. | 53.07 sec. | 62.45 sec. |
| 100 000 | 967.54 sec. | 650.85 sec. | 608.89 sec. |

TABLE III. AKKA WITH 10 ELEMENTS PER MESSAGE

| **10 elements per message** | **Active Workers** | | |
|---|---|---|---|
| **Messages processed** | ***1*** | ***10*** | ***100*** |
| 100 | 1.11 sec. | 1.38 sec. | 1.62 sec. |
| 1000 | 1.59 sec. | 1.69 sec. | 1.64 sec. |
| 10 000 | 1.56 sec. | 1.85 sec. | 2.02 sec. |
| 100 000 | 2.46 sec. | 2.58 sec. | 2.44 sec. |

TABLE IV. AKKA WITH 1 MILLION ELEMENTS PER MESSAGE

| **1 million el. per message** | **Active Workers** | | |
|---|---|---|---|
| **Messages processed** | ***1*** | ***10*** | ***100*** |
| 100 | 1.90 sec. | 1.82 sec. | 2.05 sec. |
| 1000 | 8.51 sec. | 5.69 sec. | 5.25 sec. |
| 10 000 | 69.40 sec. | 38.43 sec. | 38.25 sec. |
| 100 000 | 665.55 sec. | 356.59 sec. | 357.58 sec. |

### B. *Discussion*

Table 1 and Table 2 use only ten elements per chunk for the calculation of Pi. By doing so, the implementation is mostly busy with the message handling itself. Here ActiveMQ uses the TCP protocol to exchange the messages while Akka uses local sockets as long as messages are only exchanged on one machine and not over the network. This eliminates the need to serialize objects, which is a huge performance boost [13] [14]. Instantiating a system with only one worker and 100 000 messages, the ActiveMQ implementation takes 237.59 seconds compared to the Akka implementation which takes 2.46 seconds for the processing. Therefore, when processing a lot of messages on a local machine Akka has a huge advantage.

This advantage becomes smaller once the implementation has to calculate 1 million elements per chunk. Here the ActiveMQ implementation takes 967.54 seconds compared to Akka which takes 665.55 seconds.

Looking at Table 2 and 4 one can also see that the biggest performance improvement was gained when using ten workers instead of one. This is because the tests have been executed on a two core processor machine and by creating only one worker only one core was used for the calculation. Therefor by using more than one worker the execution time can almost be divided in half.

This measurements show that distributed systems implemented in either ActiveMQ or Akka can scale almost linear by following a scale-up approach through adding more processor cores, but are limited through the message handling. Moreover the two implementation could also follow a scale-out approach by adding additional machines and by letting the components communicate over the network, using either TCP or UDP [15] [16].

## VI. CONCLUSION

The paper started given an introduction to both toolkits, showed an implementation written in Java and tested this implementation using different parameters. The results showed that both implementation support concurrency by using multiple workers which process incoming messages in parallel. The results also show that Akka, when executed on a local machine, treats the message handling process more efficient then ActiveMQ. However, there are a lot more details to consider in order to decide if ActiveMQ or Akka should be used in order to create a solution. ActiveMQ allows using different programming languages while Akka supports only language binding for Java and Scala and must be executed in a Java Virtual Machine (JVM). Moreover, Akka and ActiveMQ can be bind together by using Akka Camel, which allows to exploit advantages of both toolkits.

## VII. FUTURE WORK

The results in the paper have been limited by the used environment. Executing the ActiveMQ and Akka implementations on a physical cluster with each machine having multiple processors, thus using a scale-out and scale-up approach, could gain further insides.

## References

[1] Java Message Service Specification. https://java.net/projects/jms-spec/pages/Home/ (Last Access on 2016-05-06)

[2] ActiveMQ Documentation. http://activemq.apache.org/ (Last Access on 2016-05-06)

[3] Akka Documentation. http://akka.io/docs/ (LastAccess on 2016-05-06)

[4] Rob Davies, Dejan Bosanac, Bruce Snyder. ActiveMQ in Action. 1st Edition, ISBN: 9781933988948

[5] Timothy Bish. Instant Apache ActiveMQ Messaging Application Development How-to. ISBN: 1782169415

[6] ActiveMQ KahaDB. http://activemq.apache.org/kahadb.html (Last Access on 2016-05-06)

[7] Edward Curry. Message-Oriented Middleware. https://www.researchgate.net/profile/Edward_Curry/publication/220035284_Message-Oriented_Middleware/links/09e41509113b28739a000000.pdf (Last Access on 2016-05-07)

[8] Overview of the JMS API. http://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html (Last Acces on 2016-05-07).

[9] Vaughn Vernon. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. 1st Edition, ISBN: 0133846830

[10] The Reactive Manifesto. http://www.reactivemanifesto.org/ (Last Access on 2016-05-07)

[11] Jonathan Borwein, David Bailey & Roland Girgensohn. Experimentation in Mathematics - Computational Paths to Discovery, ISBN 1-56881-136-5, pages 28–30.

[12] Zeng Zhe-Zhao, Wang Yao-Nan, Wen Hui. Numerical Integration Based on a Neural Network Algorithm, IEEE Computer Society July/August 2006 vol.8, pages 42-48.

[13] S Tasharofi, Stephan Rehfeld, Henrik Tramberend, Marc Erich Latoschik. An actor-based distribution model for Realtime Interactive Systems, IEEE Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2013 6th Workshop. 17-17 March 2013, pages 9 – 16

[14] Remoting. http://doc.akka.io/docs/akka/snapshot/scala/remoting.html (Last Access on 2016-05-23)

[15] ActiveMQ Transport Configuration Options. http://activemq.apache.org/configuring-transports.html (Last Access on 2016-05-23)

[16] I/O. http://doc.akka.io/docs/akka/snapshot/scala/io.html. (Last Access on 2016-05-23)

[17] Kai Sachsa, Samuel Kouneva, Jean Baconb, Alejandro Buchmanna. Performance evaluation of message-oriented middleware using the SPECjms2007, Performance Evaluation Volume 66, Issue 8, August 2009, pages 410–434

[18] Mejdi Kaddour, Laurent Pautet. A Middleware for Supporting Disconnections and Multi-Network Access in Mobile Environmen. PERCOMW, 2004, Pervasive Computing and Communications Workshops, IEEE International Conference on, Pervasive Computing and Communications Workshops, IEEE International Conference on 2004, page. 187

[19] Derek Wyatt. Let it crash. http://letitcrash.com/post/30165507578/shutdown-patterns-in-akka-2 (Last Access on 2016-05-2