Actor4j: A Software Framework for the Actor Model Focusing on the Optimization of Message Passing

David Alessandro Bauer, Juho Mäkiö
Department of Informatics and Electronics
University of Applied Sciences Emden/Leer
Emden, Germany
Email: david.bauer@hs-emden-leer.de, juho.maekioe@hs-emden-leer.de

***Abstract*—Common actor implementations often use standardized thread pools without special optimization for the message passing. For that, a high-performance solution was worked out. The actor-oriented software framework Akka uses internally a ForkJoinPool that is intended for a MapReduce approach. However, the MapReduce approach is not relevant for message passing, as it may lead to significant performance losses. One solution is to develop a thread pool that focuses on the message passing. In the implementation of the Actor4j framework, the message queue of the actors is placed in threads to enable an efficient message exchange. The actors are operated directly from this queue (injecting the message), without further ado. It was further enhanced by the use of multiple task-specific queues. Fairness and the responsiveness of the system have been raised. In particular, the performance measurement results show that an intra-thread communication towards an inter-thread communication is much better and has very good scaling properties.**

Keywords-actors; actor model; parallelization; reactive system; message passing; microservices; Java.

I. INTRODUCTION

The use of cloud computing systems is used more often, especially as a Platform-as-a-Service (PaaS) solutions (e.g., Microsoft Azure, Amazon Web Services, Google Cloud Platform). A step further is to design the architecture of software as microservices instead of a monolithic design [1]. In this case, Docker images can be used [2], which can be uploaded to them (Azure Container Service, Amazon Elastic Container Service, Google Kubernetes Engine). An alternative microservice approach is the service factory of Microsoft Azure, which orchestrates among other services. Microservices are arbitrary scalable and easy to change [3] and reusable. In Microsoft Azure service factory actors are also used (Virtual Actor pattern [4]) [5]. The advantage of actor-oriented services is that they can hold lightweight representatives (the actors). They can be used as a replacement of the traditional middle tier [4]. Actors can be seen as a pendant to Function-as-a-Service (FaaS), if the actors are themselves stateless. Actors and functions can be called nanoservices, as a lightweight derivative to microservices. Scalability can be obtained by high parallelism (to divide a task in subtasks, or parallel execution of a task, if the underlying code is stateless). See also the Scale Cube by Abbott [6], which describes the three dimensions of scalability.

To ensure high parallelization, its one benefit to use multi-core systems. The computer world of the last few years has been characterized by a change ("The Free Lunch Is Over" [7]) from constantly increasing computing power to multi-core systems due to technical limitations. In particular, technical progress always lags behind practical requirements (Wirth's law [8]). Up to now, Moore's law was "that the number of transistors available to semiconductor manufacturers would double approximately every 18 to 24 months" [9]. This will presumably continue through the transition to multi-core systems. Due to the issues with parallel programs [10] according to the classic model (especially error prone in programming of complex systems with semaphores and mutexes), actor-oriented frameworks are becoming increasingly popular [11].

This work contributes to achieving a higher performance in the field of message passing. This is relevant for all systems, where a lot of messages have to be exchanged (e.g., Internet of Things, Internet of Services). It is intended to develop a specially designed thread pool for message passing. The framework Akka is used as a reference implementation, but this is written in Scala. In order to achieve comparability and to provide a realistic picture (for benchmarks in Section 7), the degree of implementation of the underlying actor model (actor4j) must have some complexity. The four semantic properties [11] of the actor model have to be taken into account during implementation. In addition, the actor model as a reactive system should satisfy the four principles of the reactive manifesto [12]. In particular, the responsiveness of the reactive system has to be taken into account, since this is also important in regard to the achievement of this work. Akka is currently a widespread and popular (has a very good rating on GitHub [13] and a lot of contributors) actor implementation. The users of Akka are large companies like Intel, Samsung LinkedIn, Twitter and Zalando [14]. According to Suereth: "Akka is the most powerful performing framework available on the JVM" [15]. The long-term goal is the establishment of a Java framework for the actor model as an alternative to Akka. Akka is written in Scala (except the Java-API), this can be a hindrance for Java developers to understand the

 ISBN: 978-1-61208-650-7

underlying architecture. There can be also an acceptance problem.

First, two important basic building blocks of this work are discussed. Accordingly, a brief introduction to the actor model and reactive systems is given. Then, a comparison between Akka and actor4j will be presented. Next, the solution approach of the novel framework actor4j is shown. Subsequently the results of testing actor4j are presented and discussed. This paper ends up with a conclusion and insight in the future works. The source code for actor4j is available under GitHub [16].

II. ACTOR MODEL

In classic concurrent programming, it goes over the concepts of shared state, mutual exclusion and semaphores. [17]. With increasing program complexity, the correctness of the program is difficult to proof or to verify. Especially, because concurrent programs are difficult to test [18]. However, the actor model, based on message passing, offers an alternative. The essential features compared to the classic concurrent programming are:

- no shared state,
- asynchronous message transfer, and
- message queue for each actor [17].

This eliminates the need to use proprietary synchronization techniques between the actors to protect the access to shared resources [17]. The data transmitted between the actors is conceptually immutable and thus does not require synchronization [17].

When a message arrives, actors can react by:

- myself send messages,
- instantiating more actors, or
- changing its own state [19].

These activities may influence the next incoming messages (possible behavioral change) [19]. The actor model was introduced in 1973 in a paper [20] by Carl Hewitt, Peter Bishop and Richard Steiger [21]. A message can contain any kind of data.

There are "four important semantic properties of actor systems: [state] encapsulation, fairness, location transparency and mobility" [11].

The *state encapsulation* ensures that no actor can directly call another actor. Secure messaging requires that messages are transmitted in the sense of call-by-value messages. However, call-by-reference is permitted in most actor frameworks. As the "deep copying is an expensive operation" [11], this criterion is not always followed in practice. Only actors can communicate with one another via messages [11].

Fairness means that all actors can be treated equally and supplied with appropriate messages. Uncooperative actors that, for example, perform active waiting, polling, or time-consuming calls are very likely to adversely affect other actors (actors are no longer operated, blocking the corresponding thread) [11].

Location transparency means that the naming is independent of its localization. The name of the actor is unambiguous and unchangeable [11].

Mobility allows the transfer of the actor to other nodes in the cluster. A distinction is made between weak and strong mobility. Weak mobility allows for the exchange of the underlying code with subsequent initialization of the context of the actor. Strong mobility includes the current context of the actor [11].

The actor model is successfully used in business, for example in WhatsApp or for RabbitMQ (implements AMQP protocol). For both you can set up publish-subscribe systems for messaging, based on the actor model. The programming language Erlang (actor-oriented programming language, see also Section 4) was used for that.

III. REACTIVE SYSTEMS

Nowadays, more and more data need to be processed in a shorter time. This is known under the term Big Data. Big Data is associated with very large amounts of data. It may be discrete or continuous data. These fall on particularly at very high frequented and interactive web services (e.g., Facebook, Google, IoT-Area). Especially, in data mining, data is evaluated in a targeted way to create value. A practical example of a use-case is the "Deutscher Wetterdienst" (Germany's National Meteorological Service) that uses Akka for parallel processing or evaluating the historical meteorological data [22]. A solution to this can be reactive systems. Reactive systems are reacting to requested requirements. Applications should be fail-safe and easily scalable [12] for security issues.

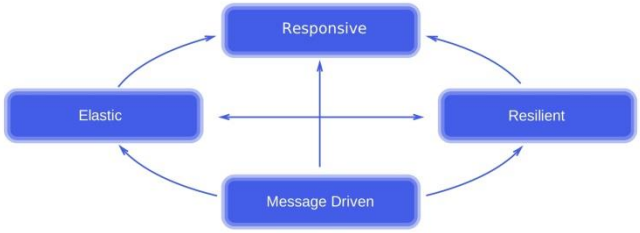

Figure 1. Presentation of the basic principles of reactive systems from the Reactive Manifesto [12]. Arrows symbolize an influence on each other.

Reactive systems are characterized by four important properties (see Figure 1):

- *Message Driven*: Messages in the reactive system are exchanged asynchronously. The components are non-transparent. [12] The actor model can serve as a basic architecture.
- *Resilient*: The reactive system is fail-safe. If errors occur, it remains responsive. Superordinate components assume the responsibility for the handling of errors (Supervision, see Erlang [23],

Akka [24]). Additional security provides the replication of functionality. [12]

- *Responsive*: The reactive system supplies time-sensitive feedback to its users and to its dependent components. This is also a prerequisite for an adequate response in the event of errors (supervision), as well as ensuring its function (task of the system). [12]
- *Elastic*: The reactive system remains adaptable even with changing requirements in regard to load capacity. If the load is changed, it can be intervened in a self-regulating manner. [12]

IV. Related Works

Akka is used as a reference implementation for Actor4j. Akka was again influenced by Erlang, in terms of fault tolerance (Supervision). The actor-oriented software framework ActorFoundry implements the four semantic properties of the actor model.

A. Erlang

Erlang is influenced by the actor model [21] and uses this directly for their language. In Erlang, so-called light-weight processes (no system processes) are used, corresponding to the actors. According to [23], the only way of communication between the processes is asynchronous message passing. Processes have a "message queue" [25] and "Processes share no Resources" [23], to eliminate the need for synchronization between the processes. The location transparency (see Section 2, Actor Model) is given by a unique process identifier (PID).

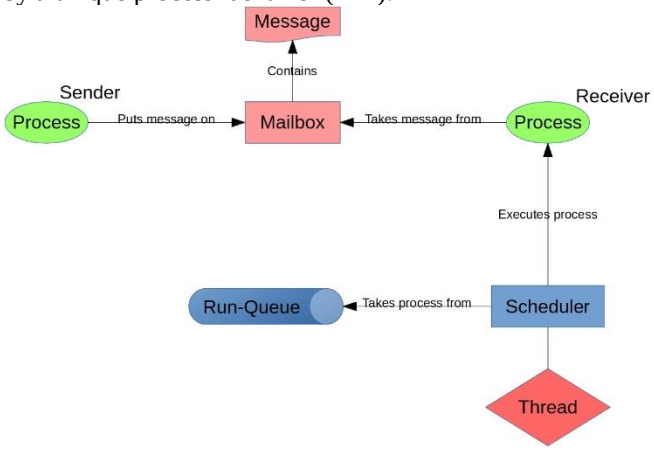

Figure 2. Schematic representation of the flow of message processing in Erlang VM (cp. [26])

Each process is assigned to a thread (1:N-architecture) and is placed in the corresponding run-queue. Processes are executed by the scheduler, that takes processes out from the run-queue (Figure 2). The process itself takes a received message out of the mailbox and processes them. Erlang can be run with one scheduler or multiple schedulers (SMP support, SMP stands for Symmetric Multiprocessing). With one scheduler synchronization is not necessary, because only one thread is interacting. The first solution for SMP was to use the schedulers with one run-queue, but this was a bottleneck. There were too much "lock conflicts" [26]. This was resolved by using one run-queue per scheduler. [26]

Characteristics of Erlang [23]:

- *Scalability*: The Erlang VM "automatically distributes the execution of processes over the available CPUs" [23]
- *Fault tolerance*: To respond adequately to faults in the processes, it is important to take precautions (see Supervision). The processes are shielded from one another so that no chain reaction occurs in the event of a failure of a process.
- *Clarity*: Processes are the representatives of a parallel reactive system. The execution of the processes runs within sequentially. Asynchronously, the exchange between the processes takes place. This structuring leads to more clarity in programming and has more reference to our real world of life.
- *Performance*: It is indisputable to see the possible performance gains when parallelizing a sequential program when this is feasible. Distributing the work to several processes can lead to success.

Supervision:

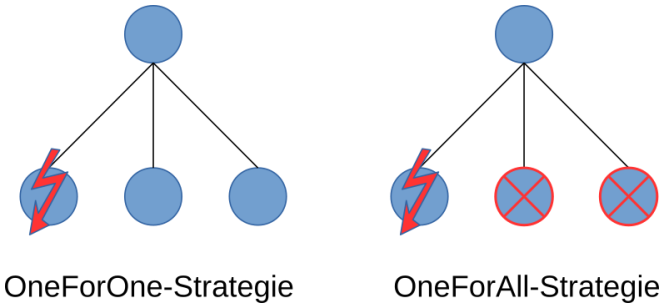

Figure 3. One-for-one supervision tree and One-for-all supervision tree [23].

The supervisor process monitors his worker processes, and in the event of an error, they are restarted. Two strategies are foreseen (see Figure 3). The OneForOne-Strategy restarts only the affected process. In the OneForAll-Strategy, on the other hand, not only the affected process is restarted, but also the neighboring processes (above the supervisor process). [23]

 ISBN: 978-1-61208-650-7

B. *Scala – Akka*

Scala is an object-functional programming language that runs on the JVM (is translated into bytecode). Since version 2.10.0, Akka is used as the default actor implementation [27]. Akka was influenced by the actor model, Erlang and Scala Actors [21]. By default, to forward messages to the actors Akka internally uses a "ForkJoinPool" from Java as a thread pool. An 1:N-architecture is used here. This means, each thread is responsible for the message delivery to its assigned actors (message is injected). The message exchange takes place via the queues of the actors.

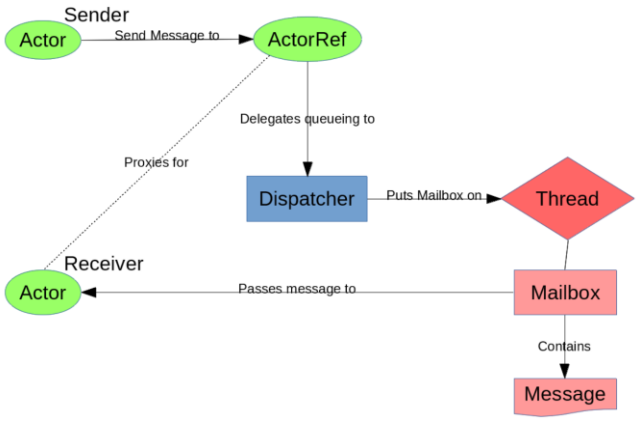

Figure 4. Schematic representation of the flow of message processing in Akka [28].

Now follows a brief explanation of the message processing in Akka. Each actor has its own mailbox (queue). The dispatcher ensures that another message is processed (Figure 4). For this purpose, a new message is taken from the mailbox of the associated actor. The message processing is executed via a thread, where a new message is injected to the actor and the message is then processed by the actor. In addition, the dispatcher ensures that no actor is called more than once at the same time.

As a thread pool, a "ForkJoinPool" is used by default. In Java 7, the "ForkJoinPool" used a central input queue for new tasks to be executed, but it was viewed as a bottleneck. With Java 8 this was improved. Instead of using a central input queue, the new task to be executed is now randomly added in one of the worker queues. „The idea is to treat external submitters in a similar way as workers - using randomized queuing and stealing" [29]. Unlike Akka, actor4j does not need these worker queues because messages are processed directly there via the message queues, belonging to the corresponding thread (see Chapter 6).

C. *ActorFoundry*

Previously, an actor was mapped to a separate thread (strict encapsulation, 1: 1-architecture). However, this led to performance problems (thread context switching). Therefore, it was switched to an 1:N-architecture. "ActorFoundry" implements the four semantic properties of the actor model adequately. Messages are transmitted by default using a "deep copy". "ActorFoundry" supports both the weak and the strong mobility. A further worker thread is provided, if uncooperative actors are recogized. This ensures system responsiveness. [11]

V. COMPARISON BETWEEN AKKA AND ACTOR4J IN TABULAR FORM

In the following section, Akka is compared with actor4j. First, the semantic properties are compared (see TABLE I). State encapsulation, fairness and location transparency were covered by both frameworks. Currently, actor4j only partly supports the mobility.

TABLE I. COMPARISON OF SEMATIC PROPERTIES BETWEEN AKKA AND ACTOR4J.

	Akka	**actor4j**
Semantic properties		
State encapsulation	Other actors cannot be referenced directly (ActorRef)	Other actors cannot be referenced directly (Universal Unique Identifier, UUID)
Fairness	Definition of a throughput	Definition of a throughput, additionally queues for different purposes
Location transparency	Actor has its unique ActorRef	Actor has its unique UUID
Mobility	Actors can be created remotely, ?	Currently partially implemented, only load balancing at creation time (related to threads)

In following, the reactiveness is compared with the reactive manifesto. Both frameworks are designed as message driven, resilient and responsive (see TABLE II). The elastic approach is currently not supported by actor4j.

TABLE II. COMPARISON OF REACTIVENESS BETWEEN AKKA AND ACTOR4J.

	Akka		**actor4j**
Reactive system			
Message driven	Asynchronous message transfer, every actor has its own message queue		Asynchronous message transfer, message queue is located at the threads
Resilient	Supervision		Supervision
Responsive	Usage of additionally thread pools		Usage of ResourceActor's for heavy computations (additionally thread pool)
Elastic	?		Currently not implemented

Both frameworks implement the following features: pattern matching, persistence, the publish-subscribe pattern, and well reactive streams (see TABLE III). Additionally,

actor4j supports an anti-flooding strategy using ring buffered queues. For enhanced performance grouping of actors is also available. Caching with actors is also supported by actor4j (volatile and persistent caching over a database).

TABLE III. COMPARISON OF ADDITIONAL FEATURES BETWEEN AKKA AND ACTOR4J.

	Akka	actor4j
Features		
	-	Anti-flooding strategy, important queues are ring buffered
	-	Grouping of actors
	Pattern matching	Pattern matching
	Persistence	Persistence
	Publish-Subscribe (see Event Bus, Event Stream)	Publish-Subscribe
	Reactive Streams	Reactive Streams
	-	Caching

For the implementation of the remote communication between actors, both frameworks use different approaches (see TABLE IV). For actor4j, applications are provided that can include several actors, which can be deployed separately into the actor system. This can ensure a domain specific separation of concerns. Akka supports failure detector, sharing and a kind of distributed publish-subscribe.

TABLE IV. COMPARISON OF CLUSTER FEATURES BETWEEN AKKA AND ACTOR4J.

	Akka	actor4j
Cluster		
	TCP, UDP, Apache Camel	REST-API, Websocket, gRPC
	-	ActorApplication planned, running in the context of an actor system
	Failure Detector	Failure Detector planned
	Sharding	Sharding planned
	Distributed Publish-Subscribe	?

For testing Akka supports (see TABLE V) behavior testing and integration testing. Actor4j supports behavior testing and a verification method, integration testing is planned.

TABLE V. COMPARISON OF TESTING FEATURES BETWEEN AKKA AND ACTOR4J.

	Akka	actor4j
Testing		
	Behaviour Testing for an actor	Behaviour Testing for an actor
	Integration Testing with JavaTestKit	Integration Testing planned
	-	Verification

VI. ACTOR4J – FINAL DESIGN

In this Section the novel thread pool architecture (see Figure 5) for actor4j is presented. Actor4j uses mainly data structures that are lock-free ("synchronized by using a lock-free technique" [30]). Therefore, in contrast to classical synchronization techniques, performance loses are avoided. With the use of lock-free programming, performance loses are possible, too. This is the case especially if multiple threads are frequently accessing the same resource (e.g., compare-and-swap conflicts).

The actor-oriented implementations presented in related works use a sort of worker-queue for the thread pooling and every actor has its own queue. The first idea was to avoid this double queuing. Now the actors belonging to the thread, will be operated directly from the thread message queue. One advantage is that actor-context switches are avoided, that would happen in the classical approach, where an access to the actors queue is needed (pushing a new message to the queue). Instead new messages are pushed to the thread message queue, avoiding the actor context at first. The disadvantage is that concurrent access (mainly inbound) conflicts are raised on the thread message queue, caused by other threads. The second idea is that actors belonging to the same thread can communicate or share resources without synchronization techniques (also absence of lock-free programming). For this, a normal (not thread-safe) queue has been set up. The third idea is to use two-level queues, one that is thread safe and one that is not. This should reduce concurrent access conflicts, from the belonging working thread. The queue to the outside is protected, the inner queues enables a higher performance in the absence of additional protecting mechanism. The two-level queues where inspired by the CPU cache levels. There was taken for the overall design the same strategy as mentioned in [26]: "First make it work, then measure, then optimize". Further explanations follow in the sub-sections below.

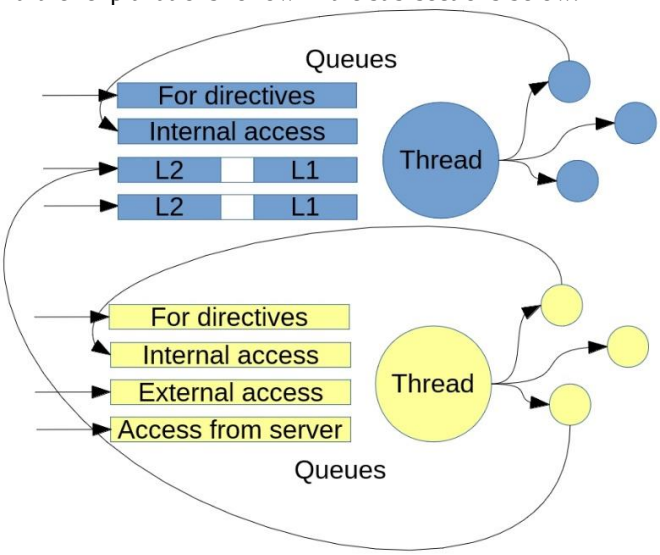

Figure 5. Presentation of the flow of message passing at actor4j (Thread pool architecture of actor4j).

A. 1:N-architecture

All actors are permanently assigned to one thread (1:N-architecture). The Thread is, in case of message delivery, responsible for injection the message and the execution of its associated actors. Actors can send messages to other actors. These messages are stored in the respective thread that is responsible for the receiving actor (see Figure 5 and Figure 6). For clarification, actors don't have their own queue, as in the classic approach.

B. Queues

The division into different queues ensures a fair message flow. This ensures that other queues are processed (whenever the thread gets a time quantum), even when the input queue is used intensively. In each round (loop within the thread) of all queues, a fixed number of messages is processed if available. This is similar to the definition of throughput in Akka [31]. So, the reactive system remains responsive.

C. Three different ways of access

All queues use a ring buffer (also an effective block for an anti-flooding strategy). If both or more corresponding actors are assigned to the same thread, the internal queue can be accessed. This is implemented as a "CircularFifoQueue" [32] because no synchronization is required in this case. If accessed from another thread, the message is placed in the external queue. This must be thread-safe now(non-blocking programming). For external access and access from the server the queue is divided into two stages.

D. Two stage division

L2 (Level 2) corresponds to a "MpscArrayQueue" [33] and L1 (Level 1) of an "ArrayDeque". This approach is intended to achieve a performance enhancement when a higher load of messages occur. The responsible thread then works mainly with L1 and loads messages accordingly. This concurrent access can be avoided to L2.

E. Directives queue

In regard to failure safety, there is also a special queue which directives are processed by the respective thread with the highest priority in order to ensure the consistency of the actor system. There are stop and restart directives that can affect single or multiple actors. If there are currently no messages at the respective thread, the thread either goes into the idle state for a short time interval or signals a yield ("Thread voluntarily releases its computing time" [34], translation).

F. Source code examples

Now follow some excerpts of the source code for clarification:

- Processing a maximum specified number of messages (throughput) per loop pass on the example of the internal queue.

```
for (; hasNextInner<system.throughput &&
     poll(innerQueue);
     hasNextInner++);
```

- For the external queue first tried L1 is to be processed. If there are no messages in L1, it will be loaded accordingly from L2.

```
for (; hasNextOuter<system.throughput &&
     poll(outerQueueL1);
     hasNextOuter++);

if (hasNextOuter<system.throughput &&
     outerQueueL2.peek()!=null) {
     ActorMessage<?> message = null;
     for (int j=0;
         j<system.getBufferQueueSize() &&
         (message=outerQueueL2.poll())!=null; j++)
             outerQueueL1.offer(message);

         for (; hasNextOuter<system.throughput &&
             poll(outerQueueL1);
             hasNextOuter++);
}
```

A complete implementation of the class ActorThread is given by default by the class DefaultActorThread [35].

G. Message processing in actor4j

Internally, message processing takes place in actor4j (see Figure 6), similar to Akka. An actor wants to send a message to another actor. This is first redirected to the corresponding ActorCell. The ActorCell class contains the actual background implementation of an actor. Each ActorCell is assigned an actor. The message is then forwarded to the dispatcher. This inserts the message according to the selected recipient into the corresponding queue of the thread (applied access options see Figure 5). As soon as the message can be processed by the thread, it is injected into the receiver actor for processing.

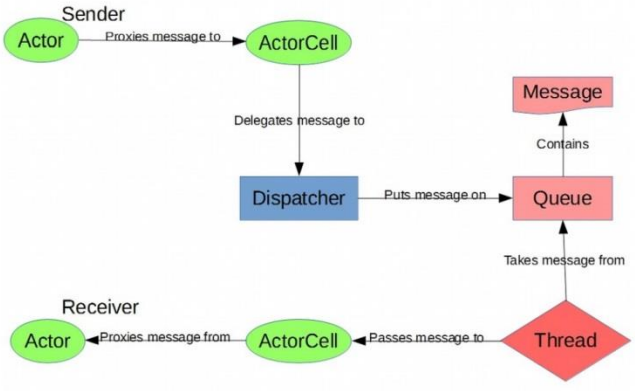

Figure 6. Schematic representation of the flow of message processing in actor4j (cp. [28] of Akka).

VII. ACTOR4J– RESULTS

The performance of message passing was tested with a DELL OptiPlex 7040, Intel® Core™ i7-6700 CPU (Skylake)

@3,40 GHz, 32 GB RAM and 8 MB L3 Cache. As the JVM Oracle JDK 9.0.4 was used under Windows 10, 64 Bit. Three benchmark scenarios are presented to get a picture of the performance of actor4j's message throughput.

1. In the first case, only the internal queue is claimed. The exchange of messages takes place on the same thread (best performance is awaited).
2. In the second case, if only the external queue is claimed. The exchange of messages takes place on different threads (worst performance is awaited).
3. As a third case, if the internal and external queue of the threads are used quasi evenly (bulk version). The exchange of messages takes place on the same or on a different thread (average performance is awaited).

Additionally, in this paper the skynet [36] benchmark as fourth benchmark is included, that shows message passing in combination of massively dynamic creating and stopping actors. In [37] this is done for "revealing the overhead for actor creation" (similar approach).

The legend "…actor4j_100" or "…akka_100" in the benchmark results means that a throughput of maximum 100 was set (cp. legends of Figure 7, Figure 8 and Figure 9). Accordingly, maximal one hundred messages per queue will be processed at once.

A. *N-fold ring benchmark*

The first is the N-fold ring benchmark. Ring or multi-ring benchmarks for actors can be found also in [11] and [37]. The idea is to bundle actors into groups, where they are guaranteed to run on the same thread and therefore no synchronization is required. For this purpose, an eightfold ring (see Figure 7) was generated for the benchmark, i.e., one ring per thread in the parallel version. Thus, no message exchange is needed between the threads at actor4j. In Akka this possibility does not exist. In the case of actor4j, only the internal queue (CircularFifoQueue) is used, since the members of the ring groups remain together on a thread. All actors are derived here from the ActorGroupMember class. In this case, Akka has no chance to equal actor4j.

In part, actor4j has a factor seven higher throughput compared to Akka. With ongoing number of actors deployed, the actor-context switches are increased in the corresponding thread. This results in less throughput. It also must be considered, that with enabled Hyper-Threading (HT), additional logical kernels through HT do not correspond to fully-fledged pure physical cores (only 30% increase in performance is expected) [9]. But with pure physical cores the result of this benchmark for actor4j should scale nearly linear, with an increased amount of cores (by less deployed actors).

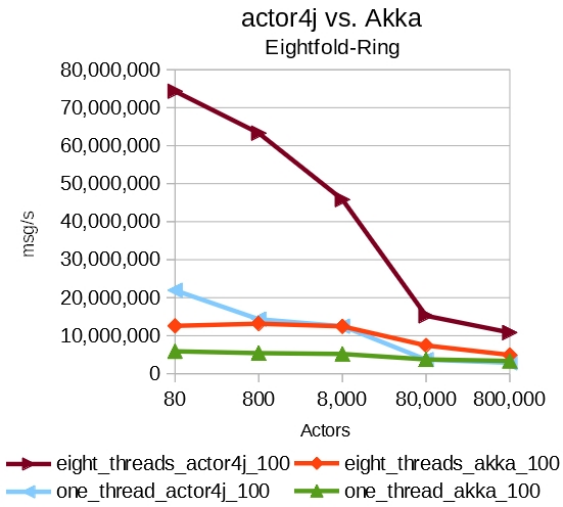

Figure 7. Results for the Eightfold-Ring benchmark. actor4j vs. Akka.

B. *Ping-Pong-Grouped benchmark*

Next, the pairs were distributed over the threads so that both partners are on a different thread. This ensures that only the external queues of the threads are used. This is only possible with actor4j in such differentiated manner. The results in Figure 8 demonstrate that actor4j has lost in performance through the intercommunication between the threads. Akka stays nearly unchanged in throughput, what was also the case in the benchmark before. Hand in hand with more actors deployed, actor-context switches reduce the message throughput. As mentioned before Akka does double queuing on the thread pool and on the actors, which is also a possible performance obstacle (see Section 4). When multithreaded and with less actors deployed, actor4j has a possible break-in in throughput, due to less work for the corresponding threads, resulting in a blocking state.

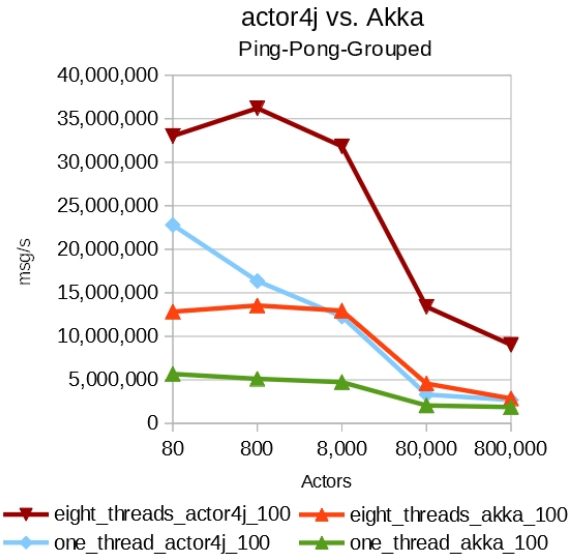

Figure 8. Results for the Ping-Pong-Grouped benchmark. actor4j vs. Akka.

 ISBN: 978-1-61208-650-7

C. Ping-Pong-Bulk benchmark

In the third benchmark (see Figure 9), the actors communicate with each other in pairs (ping-pong). In actor4j, the actors are randomly distributed over the threads. This means that both the internal faster queue (Circular FifoQueue) and the external queue (MpscArrayQueue) are used (see also Chapter 4, Actor4j-Final Design). However, the results should be interpreted with caution. Due to the random distribution of the actors, fluctuations can be expected when the benchmark is repeated. The same message is sent several times (in this case one hundred times) to the respective partner of the pairing (ping-pong), which starts the ping-pong scheme. As a result, a hundred messages are exchanged within the pairings each time the game is interchanged. This was also the benchmark for Akka or Akka.NET, with the advertising (50 million msg / s) over the resulting message throughput has been made [38].

It should be noted that Akka performs much better in bulk operations, with respect to message throughput. Both frameworks perform nearly constantly with the same throughput, for each data series. The reason for this is, that there are less actor-context switches, because a bulk operation is performed. Akka has at the last measuring point, problems to handle the massive amount of receiving messages, and struggles on that. Actor4j instead is protected by established ring buffer queues to the outside, this protects effectively against message flooding. The disadvantage of that is possibly losing messages (counteract by increasing the [buffer] queue size).

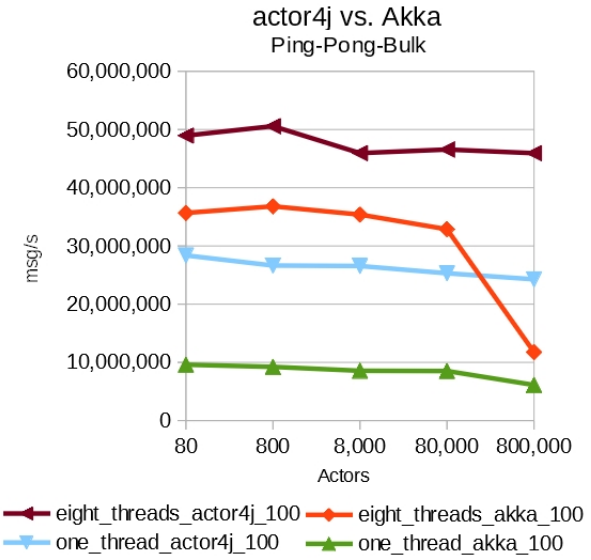

Figure 9. Results for the Ping-Pong-Bulk benchmark. actor4j vs. Akka.

Which stands out in the three benchmarks discussed before, that actor4j with one active thread reaches minimum the same (or nearly the same) message throughput as the active multi-threaded variant of Akka. At some points it has even higher message throughput.

D. Skynet benchmark

At the last benchmark [36][39], slightly over one million actors are created (exactly 1,111,111 actors), by spawning for every actor recursively tens of them. The actors are sending their ordinal number back to the parent, which are then summed up (by one million actors is this 0.5M *(1M+1)-1M), with the result of 499,999,500,000. Every branch of an actor has one child actor with the same ordinal number as his parent (so that the overall sum is correct). In Figure 10, there is an example representation of the resulting actor system structure. This benchmark can be used as a stress test, for creating and optionally stopping actors, as well that the framework is correctly implemented.

The results in TABLE VI are showing that the Akka implementation for creating and stopping actors has a better performance. For inclusively stopping actors, the Akka implementation needs three times longer than for creating them only. The reason for that is that Akka needs much more time for message passing as actor4j, as seen in the equivalent ping-pong grouped benchmark. With one thread the actor4j implementation is slightly better in performance, because of the usage of a non-synchronized queue (only in the case of non-stopping the actors). For the case "without stopping the actors", the actors will be stopped nevertheless after that, because otherwise the memory usage is going to grow constantly (is not included for calculation of the needed time).

TABLE VI. COMPARISON OF THE SKYNET BENCHMARK RESULTS.

	without stopping the actors	**without stopping the actors (only one active thread)**	**with stopping the actors**	**with stopping the actors (only one active thread)**
actor4j	5,911 ms (s=133)	4,901 ms (s=97)	8,226 ms (s=223)	9,011 ms (s=238)
Akka	2,808 ms (s=213)	3,538 ms (s=112)	7,236 ms (s=274)	8,208 ms (s=185)

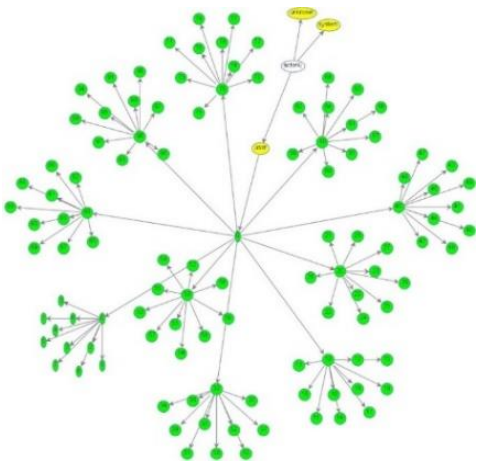

Figure 10. Structure of the actor system (Skynet benchmark with 111 actors created)

VIII. CONCLUSION

The results show that actor4j makes a more powerful impression than Akka. The final design (see Section 4) has proved to be elastic, responsive, and resilient. Actor4j was always convincing, no matter what the actor constellation (N-fold ring, ping-pong grouped, ping-pong bulk) was. These benchmarks can be used to determine the performance of inter- and intra-communication between the threads. It has been found that even with the use of lock-free queues these counter against a good scaling (see the ping-pong benchmark). On the other hand, a very good scaling is obtained with intra-communication, i.e., within a thread.

A. Advice

It is advisable to keep communication-active actors together in one and on the same thread, especially if they have a bounded context (see Domain Driven Design). By bounded context is meant that an assignment to a together interacting actor system is possible. With the actor model, agent-based systems can be implemented well. For example, it is useful to keep the ant grouped together as an actor system (sensors, actuators, control unit) in an ant simulation. An own basic ant simulation was built with Akka. An ant is built up by a composition of systems which are describing a comparable SDA-cycle (sense-decide-act) [40]. It is expected that more interaction will occur within the ant system than the environment. In addition, scaling is easier to implement as additional actors are distributed to more threads (when more processors are used).

B. Concept of the new architecture

In the classic design, one message queue is assigned to each actor. This was relocated as already presented to the competent actors thread. In theory, that makes sense. In the real world there is a medium, the surrounding world, between two actors. In particular, the air, which transmits speech through the sound and can be recorded by an actor by its sensors. This can be transferred to the actor model. This means, it makes sense that there is a kind of network layer between the actors, which temporarily stores messages for the actors. Actor4j is also oriented on the four semantic properties of the actor model (see Section 2). With actor4j it is possible to replace very easily the default thread and dispatcher implementation. Therefore, the framework is very flexible, for changing or different providing requirements.

C. Compliance of the four semantic properties

Communication partners are awarded in actor4j via their UUID. Direct access to another actor is so avoided (*encapsulation*). By default, a "deep copy" is carried out for the message transmission, if the prerequisites are fulfilled (interface Copyable implemented for the payload). The payload contains the actual message. The header (sender, recipient, tag) of the message is copied. A new instance of ActorMessage is generated that contains the header and the payload. Senders and receivers are represented by a UUID. The UUIDs do not change (final). It is also possible to transfer the payload as call-by-reference (without "deep copy"). This remains in the responsibility of the developer.

By alternately processing the queues in the actor threads, *fairness* is given. By adjusting the value of throughput, the degree of fairness can be adjusted. A throughput of one would be absolutely fair [31], but the message processing would then be more inefficient (reduction of the message throughput). The order in which messages are transmitted within an actor thread is given (intra-communication). In inter-threading communication, the sequence is only observed between two interacting actors [41]. Otherwise, message communication is not deterministic [19].

The *location transparency* is ensured by the unique UUID for actors. In actor4j, the assignment of an alias is also possible for the simple identification of an actor. A transfer of actors within the actor system (here: relocation to other threads) is currently not implemented (also not in the cluster).

The first purpose of *mobility* is load balancing. Another reason is the displacement of actors to a different location.

D. Future work

One problem is that as the number of actors increases, the throughput drops further and further. This is caused by the constant actor-context switching. Probably this cannot be avoided unless the computing power is increased (higher clock frequency or more physical cores). One useful enhancement could be a special priority queue (belongs to the thread), for prioritized tasks, which can be added by the actors. It is planned to test the actor4j framework under the EU project STIMEY.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement Nº 709515 — STIMEY

REFERENCES

[1] Microsoft Azure, "Why a microservices approach to building applications," 2016, [Online]. Available from: https://azure.microsoft.com/engb/documentation/articles/service-fabric-overview microservices/ [retrieved: October, 2016].

[2] M. Jansen and B. Wenzel, "Microservice-Architektur mit Docker und Kubernetes," in Java mit Integrations-SPEKTRUM, Issue 1, February/March 16, pp. 8-14, 2016.

[3] Microsoft Azure, "Overview of Service Fabric," 2016, [Online]. Available from: https://azure.microsoft.com/engb/documentation/articles/service-fabric-overview/ [retrieved: October, 2016].

[4] Microsoft, "Orleans - Virtual Actors," 2010, [Online]. Available from: https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/ [retrieved: June, 2018]

[5] Microsoft Azure, "Introduction to Service Fabric Reliable Actors," 2016, [Online]. Available from: https://azure.microsoft.com/en-gb/documentation/articles/service-fabric-reliable-actors-introduction/ [retrieved: June, 2018]

 ISBN: 978-1-61208-650-7

[6] M. Abbott, “Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise,” Addison-Wesley, p. 340, 2015.

[7] H. Sutter, “The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software,” In: Dr. Dobb's Journal 30(3), 2015, [Online]. Available from: http://www.gotw.ca/publications/concurrency-ddj.htm [retrieved: June, 2018]

[8] N. Wirth, "A plea for lean software," in *Computer*, vol. 28, no. 2, pp. 64-68, Feb 1995.

[9] S. Akhter and J. Roberts, “Multi-Core Programming: Increasing Performance through Software Multi-threading,” Intel Corporation, p. 7, 2006.

[10] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar, “Permission Regions for Race-Free Parallelism”, in Khurshid S., Sen K. (eds) Runtime Verification, RV 2011. Lecture Notes in Computer Science, vol. 7186, Springer, Berlin, Heidelberg, 2012.

[11] R. K. Karmani, A. Shali, and G. Agha, “Actor frameworks for the JVM platform: a comparative analysis,” in PPPJ, ACM, 2009, [Online]. Available from: http://osl.cs.illinois.edu/media/papers/karmani-2009-pppj-actor_frameworks_for_the_jvm_platform.pdf [retrieved: June, 2018]

[12] J. Bonér, D. Farley, R. Kuhn, M. Thompson, and Community, “The Reactive Manifesto,” 2014, [Online]. Available from: http://www.reactivemanifesto.org/ [retrieved: June, 2018].

[13] Lightbend Inc, “Akka,” 2018, [Online]. Available from: https://github.com/akka/akka [retrieved: June, 2018].

[14] Lightbend Inc. “Akka: Case studies,” 2018, [Online]. Available from: https://www.lightbend.com/case-studies [retrieved: June, 2018]

[15] J. D. Suereth, “Scala in Depth,” p. 229, Manning, 2012.

[16] D. A. Bauer, “Actor4j an actor implementation,” 2017, [Online]. Available from: https://github.com/relvaner/actor4j-core [retrieved: June, 2018]

[17] A. Miller, “Understanding actor concurrency, Part 1: Actors in Erlang,” in JavaWorld, [Online]. Available from: https://www.javaworld.com/article/2077999/java-concurrency/understanding-actor-concurrency--part-1--actors-in-erlang.html [retrieved: June, 2018]

[18] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, “Java Concurrency in Practice,” p. 273, 2006.

[19] R. K. Karmani and G. Agha. “Actors,” in Encyclopedia of Parallel Computing, Springer, 2011, [Online]. Available from: http://osl.cs.illinois.edu/media/papers/karmani-2011-actors.pdf [retrieved: June, 2018]

[20] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular Actor formalism for artificial intelligence,” in 3rd International Joint Conference on Artificial Intelligence (IJCAI), pp. 235-245, 1973.

[21] Lightbend Inc, “Akka: Actors,” 2016, [Online]. Available from: http://doc.akka.io/docs/akka/2.4.0/java/untyped-actors.html [retrieved: June, 2018]

[22] M. Lehmann and M. Werner, “Gut Wetter machen! Java, Play und Akka für meteorologische Anwendungen beim Deutschen Wetterdienst,” in JavaSPEKTRUM 3/2016, 2016

[23] J. Armstrong, “Programming Erlang: Software for a Concurrent World (Pragmatic Programmers),” Pragmatic Bookshelf, 2013.

[24] Lightbend Inc, “Akka: Supervision and Monitoring,” 2018, [Online]. Available from: https://doc.akka.io/docs/akka/2.5/general/supervision.html [retrieved: June, 2018]

[25] J. Barklund and R. Virding, “The Erlang 4.7.3 Reference Manual,” p. 133, 1999.

[26] K. Lundin, “Inside the Erlang VM,” Ericsson AB, 2008, [Online]. Available from: http://www.erlang.se/euc/08/euc_smp.pdf [retrieved: June, 2018]

[27] V. Jovanovic and P. Haller, “The Scala Actors Migration Guide,” EPFL, [Online]. Available from: https://docs.scala-lang.org/overviews/core/actors-migration-guide.html [retrieved: June, 2018]

[28] D. Wyatt, “AKKA Concurrency,” Artima Inc, p. 72, 2013.

[29] D. Lea, “ForkJoin updates,” 2012, [Online]. Available from: http://cs.oswego.edu/pipermail/concurrency-interest/2012-January/008987.html [retrieved: June, 2018]

[30] M. Botincan and D. Runje, “Lock-Free Stack and Queue. Java vs .NET,” 29th International Conference on Information Technology Interfaces, Cavtat, pp. 741-746, 2007

[31] Lightbend Inc, “Akka: Dispatchers,” 2016, [Online]. Available from: https://doc.akka.io/docs/akka/snapshot/dispatchers.html?language=scala [retrieved: June, 2018]

[32] The Apache Software Foundation, “Apache Commons Collections,” 2014, [Online]. Available from: https://commons.apache.org/proper/commons-collections/ [retrieved: June, 2018]

[33] N. Wakart, “JCTools,” 2015, [Online]. Available from: https://github.com/JCTools/JCTools [retrieved: June, 2018]

[34] C. Ullenboom, “Java ist auch eine Insel: Das umfassende Handbuch,” Kapitel 14.3.5, Galileo Computing, 2010, [Online]. Available from: http://openbook.rheinwerk-verlag.de/javainsel9/ [retrieved: June, 2018]

[35] D. A. Bauer, “Actor4j: DefaultActorThread,” 2017, [Online]. Available from: https://github.com/relvaner/actor4j-core/blob/master/src/main/java/actor4j/core/DefaultActorThread.java [retrieved: June, 2018]

[36] A. Temerev, “Skynet 1M concurrency microbenchmark,” 2016, [Online]. Available from: https://github.com/atemerev/skynet [retrieved: June, 2018]

[37] D. Charousset, R. Hiesgen, and T. C. Schmidt, “Revisiting actor programming in C++,” in Computer Languages, Systems & Structures 45, pp. 105-131, 2016, [Online]. Available from: https://actor-framework.org/pdf/chs-rapc-16.pdf [retrieved: June, 2018]

[38] Lightbend Inc, “Akka: TellThroughputPerformanceSpec.scala”, 2014, [Online]. Available from: https://github.com/akka/akka/blob/release-2.3/akka-actor-tests/src/test/scala/akka/performance/microbench/TellThroughputPerformanceSpec.scala [retrieved: June, 2018]

[39] R. Pressler, “Go and Quasar: A Comparison of Style and Performance,” in DZone, 2016, [Online]. Available from: https://dzone.com/articles/go-and-quasar-a-comparison-of-style-and-performanc [retrievd: June, 2018].

[40] M. Wooldrige, “An Inroduction to MultiAgent Systems,” John Wiley & Sons Ltd, p. 22, 2009.

[41] Lightbend Inc, “Akka: Message Delivery Reliability. Discussion: Message Ordering,” 2015, [Online]. Available from: http://doc.akka.io/docs/akka/2.4.1/general/message-delivery-reliability.html# message-ordering [retrieved: June, 2018]