Programming with actors in Java 8*

Behrooz Nobakht[1,2] and Frank S. de Boer[3]

[1] Leiden Advanced Institute of Computer Science
Leiden University
bnobakht@liacs.nl
[2] SDL Fredhopper
bnobakht@sdl.com
[3] Centrum Wiskunde en Informatica
frb@cwi.nl

Abstract. There exist numerous languages and frameworks that support an implementation of a variety of actor-based programming models in Java using concurrency utilities and threads. Java 8 is released with fundamental new features: lambda expressions and further dynamic invocation support. We show in this paper that such features in Java 8 allow for a high-level actor-based methodology for programming distributed systems which supports the programming to interfaces discipline. The embedding of our actor-based Java API is shallow in the sense that it abstracts from the actual thread-based deployment models. We further discuss different concurrent execution and thread-based deployment models and an extension of the API for its actual parallel and distributed implementation. We present briefly the results of a set of experiments which provide evidence of the potential impact of lambda expressions in Java 8 regarding the adoption of the actor concurrency model in large-scale distributed applications.

Keywords: Actor model, Concurrency, Asynchronous Message, Java, Lambda Expression

1 Introduction

Java is beyond doubt one of the mainstream object oriented programming languages that supports a *programming to interfaces* discipline [9,35]. Through the years, Java has evolved from a mere programming language to a huge platform to drive and envision standards for mission-critical business applications. Moreover, the Java language itself has evolved in these years to support its community with new language features and standards. One of the noticeable domains of focus in the past decade has been distribution and concurrency in research and application. This has led to valuable research results and numerous libraries and frameworks with an attempt to provide distribution and

* This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, http://www.envisage-project.eu.

concurrency at the level of Java language. However, it is widely recognized that the thread-based model of concurrency in Java that is a well-known approach is not appropriate for realizing distributed systems because of its inherent synchronous communication model. On the other hand, the event-driven actor model of concurrency introduced by Hewitt [17] is a powerful concept for modeling distributed and concurrent systems [2,1]. Different extensions of actors are proposed in several domains and are claimed to be the most suitable model of computation for many applications [18]. Examples of these domains include designing embedded systems [25,24], wireless sensor networks [6], multi-core programming [22] and delivering cloud services through SaaS or PaaS [5]. This model of concurrent computation forms the basis of the programming languages Erlang [3] and Scala [16] that have recently gained in popularity, in part due to their support for scalable concurrency. Moreover, based on the Java language itself, there are numerous libraries that provide an implementation of an actor-based programming model.

The main problem addressed in this paper is that in general existing actor-based programming techniques are based on an explicit encoding of mechanisms at the application level for message passing and handling, and as such overwrite the general object-oriented approach of method look-ups that forms the basis of programming to interfaces and the design-by-contract discipline [26]. The entanglement of event-driven (or asynchronous messaging) and object-oriented method look-up makes actor-based programs developed using such techniques extremely difficult to reason about and formalize. This clearly hampers the promotion of actor-based programming in mainstream industry that heavily practices object-oriented software engineering.

The main result of this paper is a Java 8 API for programming distributed systems using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment) and fully supports programming to interfaces discipline. We discuss the API architecture, its properties, and different concurrent execution models for the actual implementation.

Our main approach consists of the explicit description of an actor in terms of its *interface*, the use of the recently introduced lambda expressions in Java 8 in the implementation of asynchronous message passing, and the formalization of a corresponding high-level actor programming methodology in terms of an executable modeling language which lends itself to formal analysis, ABS [20].

The paper continues as follows: in Section 2, we briefly discuss a set of related works on actors and concurrent models especially on JVM platform. Section 3 presents an example that we use throughout the paper, we start to model the example using a library. Section 4 briefly introduces a concurrent modeling language and implements the example. Section 5 briefly discusses Java 8 features that this works uses for implementation. Section 6 presents how an actor model maps into programming in Java 8. Section 7 discusses in detail the implementation architecture of the actor API. Section 8 discusses how a number of benchmarks were performed for the implementation of the API and how

they compare with current related works. Section 9 concludes the paper and discusses the future work.

2 Related Work

There are numerous works of research and development in the domain of actor modeling and implementation in different languages. We discuss a subset of the related work in the level of modeling and implementation with more focus on Java and JVM-based efforts in this section.

Erlang [3] is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks. Elixir [33] is a functional meta-programming aware language built on top of the Erlang VM. It is a dynamic language with flexible syntax with macros support that leverages Erlang's abilities to build concurrent, distributed, fault-tolerant applications with hot code upgrades.

Scala is a hybrid object-oriented and functional programming language inspired by Java. The most important concept introduced in [16] is that Scala actors unify *thread-based* and *event-based* programming model to fill the gap for concurrency programming. Through the event-based model, Scala also provides the notion of continuations. Scala provides quite the same features of scheduling of tasks as in concurrent Java; i.e., it does not provide a direct and customizable platform to manage and schedule priorities on messages sent to other actors. Akka [15] is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

Kilim [31] is a message-passing framework for Java that provides ultra-lightweight threads and facilities for fast, safe, zero-copy messaging between these threads. It consists of a bytecode postprocessor (a "weaver"), a run time library with buffered mailboxes (multi-producer, single consumer queues) and a user-level scheduler and a type system that puts certain constraints on pointer aliasing within messages to ensure interference-freedom between threads. The SALSA [34,22] programming language (Simple Actor Language System and Architecture) is an active object-oriented programming language that uses concurrency primitives beyond asynchronous message passing, including token-passing, join, and first-class continuations.

RxJava [7] by Netflix is an implementation of reactive extensions [27] from Microsoft. Reactive extensions try to provide a solution for composing asynchronous and event-based software using observable pattern and scheduling.

An interesting direction of this library is that it uses reactive programming to avoid a phenomenon known as "callback hell"; a situation that is a natural consequence of composing `Future` abstractions in Java specifically when they wait for one another. However, RxJava advocates the use of asynchronous functions that are triggered in response to the other functions. In the same direction, LMAX Disruptor [4,8] is a highly concurrent event processing framework that takes the approach of event-driven programming towards provision of concurrency and asynchronous event handling. The system is built on the JVM platform and centers on a Business Logic Processor that can handle 6 million events per second on a single thread. The Business Logic Processor runs entirely in-memory using event sourcing. The Business Logic Processor is surrounded by Disruptors - a concurrency component that implements a network of queues that operate without needing locks.

3 State of the Art: An example

In the following, we illustrate the state of the art in actor programming by means of a simple example using the Akka [32] library which features asynchronous messaging and which is used to program actors in both Scala and Java. We want to model in Akka an "asynchronous ping-pong match" between two actors represented by the two interfaces `IPing` and `IPong` which are depicted in Listings 1 and 2. An asynchronous call by the actor implementing the `IPong` interface of the `ping` method of the actor implementing the `IPing` interface should generate an asynchronous call of the `pong` method of the callee, and vice versa. We intentionally design `ping` and `pong` methods to take arguments in order to demonstrate how method arguments may affect the use of an actor model in an object-oriented style.

Listing 1: Ping as an interface

```
1 public interface IPing {
2   void ping(String msg);
3 }
```

Listing 2: Pong as an interface

```
1 public interface IPong {
2   void pong(String msg);
3 }
```

To model an actor in Akka by a class, say `Ping`, with interface `IPing`, this class is required *both* to *extend* a given pre-defined class `UntypedActor` and *implement* the interface `IPing`, as depicted in Listings 3 and 4. The class `UntypedActor` provides two Akka framework methods `tell` and `onReceive` which are used to enqueue and dequeue asynchronous messages. An asynchronous call to, for example, the method `ping` then can be modeled by passing a user-defined encoding of this call, in this case by prefixing the string argument with the string "pinged", to a (synchronous) call of the `tell` method which results in enqueuing the message. In case this message is dequeued the implementation of the `onReceive` method as provided by the `Ping` class then calls the `ping` method.

Listing 3: Ping actor in Akka

```
1  public class Ping(ActorRef pong)
2    extends UntypedActor
3    implements IPing {
4
5    public void ping(String msg) {
6      pong.tell("ponged," + msg)
7    }
8
9    public void onReceive(Object m)
         {
10     if (!(m instanceof String)) {
11       // Message not understood.
12     } else
13     if (((String) m).startsWith("
           pinged") {
14       // Explicit cast needed.
15       ping((String) m);
16     }
17   }
18 }
```

Listing 4: Pong class in Akka

```
1  public class Pong
2    extends UntypedActor
3    implements IPong {
4
5    public void pong(String msg) {
6      sender().tell(
7        "pinged," + msg);
8    }
9
10   public void onReceive(Object m)
         {
11     if (!(m instanceof String)) {
12       // Message not understood.
13     } else
14     if (m.startsWith("ponged") {
15       // Explicit cast needed.
16       ping((String) m);
17     }
18   }
19 }
```

Access to the sender of the message in Akka is provided by `sender()`. In the main method as described in Listing 5 we show how the initialize and start the ping/pong match. Note that a reference to the "pong" actor is passed to the "ping" actor.

Listing 5: main in Akka

```
1 ActorSystem s = ActorSystem.create
      ();
2 ActorRef pong = s.actorOf(Props.
      create(Pong.class));
3 ActorRef ping = s.actorOf(Props.
      create(Ping.class, pong));
4 ping.tell(""); // To get a Future
```

Further, both the `onReceive` methods are invoked by Akka `ActorSystem` itself. In general, Akka actors are of type `ActorRef` which is an abstraction provided by Akka to allow actors send asynchronous messages to one another. An immediate consequence of the above use of inheritance is that the class `Ping` is now exposing a public behavior that is *not* specified by its *interface*. Furthermore, a "ping" object refers to a "pong" object by the type `ActorRef` . This means that the interface `IPong` is not directly visible to the "ping" actor. Additionally, the implementation details of receiving a message should be "hand coded" by the programmer into the special method `onReceive` to define the responses to the received messages. In our case, this implementation consists of a decoding of the message (using type-checking) in order to *look up* the method that subsequently should be invoked. This fundamentally interferes with the general object-oriented mechanism for method look-up which forms the basis of the programming to interfaces discipline. In the next section, we continue the same example and discuss an actor API for directly calling asynchronously methods using the general object-oriented mechanism for method look-up. Akka has recently released

a new version that supports Java 8 features [4]. However, the new features can be categorized as syntax sugar on how incoming messages are filtered through object/class matchers to find the proper type.

4 Actor Programming in Java

We first describe informally the actor programming model assumed in this paper. This model is based on the Abstract Behavioral Specification language (ABS) introduced in [20]. ABS uses asynchronous method calls, futures, interfaces for encapsulation, and cooperative scheduling of method invocations inside concurrent (active) objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. More specifically, actors in ABS have an identity and behave as active objects with encapsulated data and methods which represent their state and behavior, respectively. Actors are the units of concurrency: conceptually an actor has a dedicated processor. Actors can only send asynchronous messages and have queues for receiving messages. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. A method is a piece of sequential code that may send messages. Asynchronous method calls use futures as dynamically generated references to return values. The execution of a method can be (temporarily) suspended by release statements which give rise to a form of cooperative scheduling of method invocations inside concurrent (active) objects. Release statements can be conditional (e.g., checking a future for the return value). Listings 7, 8 and 6 present an implementation of ping-pong example in ABS. By means of the statement on line 6 of Listing 7 a "ping" object directly calls asynchronously the `pong` method of its "pong" object, and vice versa. Such a call is stored in the message queue and the called method is executed when the message is dequeued. Note that variables in ABS are declared by interfaces. In ABS, `Unit` is similar to `void` in Java.

Listing 6: main in ABS

```
1 ABSIPong pong;
2 pong = new ABSPong;
3 ping = new ABSPing(pong);
4 ping ! ping("");
```

[4] Documentation available at `http://doc.akka.io/docs/akka/2.3.2/java/lambda-index-actors.html`

Listing 7: Ping in ABS

```
1 interface ABSIPing {
2   Unit ping(String msg);
3 }
4 class ABSPing(ABSIPong pong)
      implements ABSIPing {
5   Unit ping(String msg) {
6   pong ! pong("ponged," + msg);
7   }
8 }
```

Listing 8: Pong in ABS

```
1 interface ABSIPong {
2   Unit pong(String msg);
3 }
4 class ABSPong implements ABSIPong
      {
5   Unit pong(String msg) {
6     sender ! ping( "pinged," + msg
          );
7   }
8 }
```

5 Java 8 Features

In the next section, we describe how ABS actors are implemented in Java 8 as API. In this section we provide an overview of the features in Java 8 that facilitate an efficient, expressive, and precise implementation of an actor model in ABS.

Java Defender Methods Java *defender* methods (JSR 335 [13]) use the new keyword **`default`**. Defender methods are declared for **`interface`**s in Java. In contrast to the other methods of an interface, a default method is not an abstract method but must have an implementation. From the perspective of a client of the interface, defender methods are no different from ordinary interface methods. From the perspective of a hierarchy descendant, an implementing class can optionally *override* a default method and change the behavior. It is left as a decision to any class implementing the interface whether or not to override the default implementation. For instance, in Java 8 `java.util.Comparator` provides a default method `reversed()` that creates a reversed-order comparator of the original one. Such default method eliminates the need for any implementing class to provide such behavior by inheritance.

Java Functional Interfaces Functional interfaces and lambda expressions (JSR 335 [13]) are fundamental changes in Java 8. A **`@FunctionalInterface`** is an annotation that can be used for interfaces in Java. Conceptually, any class or interface is a functional interface if it consists of exactly one *abstract* method. A lambda expression in Java 8, is a runtime translation [11] of any type that is replaceable by a functional interface. Many of Java's classic interfaces are functional interfaces from the perspective of Java 8 and can be turned into lambda expressions; e.g. `java.lang.Runnable` or `java.util.Comparator`. For instance,

```
(s1, s2) → return s1.compareTo(s2);
```

is a lambda expression that can be statically cast to an instance of a `Comparator<String>`; because it can be replaced with a functional interface that has a method with two strings and returning one integer. Lambda expressions in Java 8 *do not* have an intrinsic type. Their type is bound to the context that they are used in but

their type is always a functional interface. For instance, the above definition of a lambda expression can be used as:

```
Comparator<String> cmp1 = (s1, s2) → return s1.compareTo(s2);
```

in one context while in the other:

```
Function<String> cmp2 = (s1, s2) → return s1.compareTo(s2);
```

given that `Function<T>` is defined as:

```
interface Function<T> { int apply(T t1, T t2); }
```

In the above examples, the same lambda expression is statically cast to a different matching functional interface based on the context. This is a fundamental new feature in Java 8 that facilitates application of functional programming paradigm in an object-oriented language.

This work of research extensively uses this feature of Java 8. Java 8 marks many of its own APIs as functional interfaces most important of which in this context are `java.lang.Runnable` and `java.util.concurrent.Callable`. This means that a lambda expression can replace an instance of `Runnable` or `Callable` at runtime by JVM. We will discuss later how we utilize this feature to allow us model an asynchronous message into an instance of a `Runnable` or `Callable` as a form of a lambda expression. A lambda expression equivalent of a `Runnable` or a `Callable` can be treated as a queued message of an actor and executed.

Java Dynamic Invocation Dynamic invocation and execution with method handles (JSR 292 [29]) enables JVM to support efficient and flexible execution of method invocations in the absence of static type information. JSR 292 introduces a new byte code instruction `invokedynamic` for JVM that is available as an API through `java.lang.invoke.MethodHandles`. This API allows translation of lambda expression in Java 8 at runtime to be executed by JVM. In Java 8, use of lambda expression are favored over anonymous inner classes mainly because of their performance issues [12]. The abstractions introduced in JSR 292 perform better that Java Reflection API using the new byte code instruction. Thus, lambda expressions are compiled and translated into method handle invocations rather reflective code or anonymous inner classes. This feature of Java 8 is indirectly use in ABS API through the extensive use of lambda expressions. Moreover, in terms of performance, it has been revealed that invoke dynamic is much better than using anonymous inner classes [12].

6 Modeling actors in Java 8

In this section, we discuss how we model ABS actors using Java 8 features. In this mapping, we demonstrate how new features of Java 8 are used.

The `Actor` Interface We introduce an interface to model actors using Java 8 features discussed in Section 5. Implementing an interface in Java means that the object exposes public APIs specified by the interface that is considered the behavior of the object. Interface implementation is opposed to inheritance extension in which the object is possibly forced to expose behavior that may not be part of its intended interface. Using an interface for an actor allows an object to preserve its own interfaces, and second, it allows for multiple interfaces to be implemented and composed.

A Java API for the implementation of ABS models should have the following main three features. First, an object should be able to send asynchronously an arbitrary message in terms of a method invocation to a receiver actor object. Second, sending a message can optionally generate a so-called future which is used to refer to the return value. Third, an object during the processing of a message should be able to access the "sender" of a message such that it can reply to the message by another message. All the above must co-exist with the fundamental requirement that for an object to act like an actor (in an object-oriented context) should *not* require a modification of its intended interface.

The `Actor` interface (Listings 9 and 10) provides a set of **`default`** methods, namely the `run` and `send` methods, which the implementing classes do not need to re-implement. This interface further encapsulates a queue of messages that supports concurrent features of Java API [5]. We distinguish two types of messages: messages that are not expected to generate any result and messages that are expected to generate a result captured by a future value; i.e. an instance of `Future` in Java 8. The first kind of messages are modeled as instances of `Runnable` and the second kind are modeled instances of `Callable`. The default `run` method then takes a message from the queue, checks its type and executes the message correspondingly. On the other hand, the default (overloaded) `send` method stores the sent message and creates a future which is returned to the caller, in case of an instance of `Callable`.

[5] Such API includes usage of different interfaces and classes in `java.util.concurrent` package [23]. The concurrent Java API supports blocking and synchronization features in a high-level that is abstracted from the user.

Listing 9: Actor interface (1)

```
1  public interface Actor {
2    public void run() {
3      Object m = queue.take();
4
5      if (m instanceof Runnable) {
6        ((Runnable) m).run();
7      } else
8
9      if (m instanceof Callable) {
10       ((Callable) m).call();
11     }
12   }
13
14   // continue to the right
```

Listing 10: Actor interface (2)

```
1
2    public void send(Runnable m) {
3      queue.offer(m);
4    }
5
6    public <T> Future<T>
7      send(Callable<T> m) {
8        Future<T> f =
9          new FutureTask(m);
10       queue.offer(f);
11       return f;
12   }
13 }
```

Modeling Asynchronous Messages We model an asynchronous call

$$\texttt{Future<V>}\ f = e_0\ !\ m(e_1, \ldots, e_n)$$

to a method in ABS by the Java 8 code snippet of Listing 11. The final local variables $u_1, \ldots, u_n$ (of the caller) are used to store the values of the Java 8 expressions $e_1, \ldots, e_n$ corresponding to the actual parameters $e_1, \ldots, e_n$. The types T_i, $i = 1, \ldots, n$, are the corresponding Java 8 types of e_i, $i = 1, \ldots, n$.

Listing 11: Async messages with futures

```
final T1 u1 = e1;
. . .
final Tn un = en;
Future<V> v = e0.send(
  () → { return m(u1,...,un); }
);
```

Listing 12: Async messages w/o futures

```
final T1 u1 = e1;
. . .
final Tn un = en;
e0.send(
  { () → m(u1,...,un); }
);
```

The lambda expression which encloses the above method invocation is an instance of the functional interface; e.g. `Callable`. Note that the generated object which represents the lambda expression will contain the local context of the caller of the method "m" (including the local variables storing the values of the expressions $e_1, \ldots, e_n$), which will be restored upon execution of the lambda expression. Listing 12 models an asynchronous call to a method without a return value.

As an example, Listings 13 and 14 present the running ping/pong example, using the above API. The main program to use ping and pong implementation is presented in Listing 15.

Listing 13: Ping as an Actor

```
1 public class Ping(IPong pong)
       implements IPing, Actor {
2  public void ping(String msg) {
3   pong.send( () → { pong.("ponged
        ," + msg) } );
4   }
5 }
```

Listing 14: Pong as an Actor

```
1 public class Pong implements IPong
      , Actor {
2   public void pong(String msg) {
3     sender().send(( ) → { ping.("
          pinged," + msg) } );
4   }
5 }
```

As demonstrated in the above examples, the "ping" and "pong" objects *preserve* their own *interfaces* contrary to the example depicted in Section 3 in which the objects *extend* a specific "universal actor abstraction" to inherit methods and behaviors to become an actor. Further, messages are processed *generically* by the `run` method described in Listing 9. Although, in the first place, sending an asynchronous may look like to be able to change the recipient actor's state, this is not correct. The variables that can be used in a lambda expression are *effectively* final. In other words, in the context of a lambda expression, the recipient actor only provides a snapshot view of its state that cannot be changed. This prevents abuse of lambda expressions to change the receiver's state.

Modeling Cooperative Scheduling The ABS statement **`await`** g, where g is a boolean guard, allows an active object to preempt the current method and schedule another one. We model cooperative scheduling by means of a call to the `await` method in Listing 16. Note that the preempted process is thus passed as an additional parameter and as such queued in case the guard is false, otherwise it is executed. Moreover, the generation of the continuation of the process is an optimization task for the code generation process to prevent code duplication.

Listing 15: main in ABS API

```
1 IPong pong = new Pong();
2 IPing ping = new Ping(pong);
3 ping.send(
4   () -> ping.ping("")
5 );
```

Listing 16: Java 8 await implementation

```
1 void await(final Boolean guard,
2            final Runnable cont) {
3   if (!guard) {
4     this.send(() →
5       { this.await(guard, cont) })
6   } else { cont.run() }
7 }
```

7 Implementation Architecture

Figure 1 presents the general layered architecture of the actor API in Java 8. It consists of three layers: the routing layer which forms the foundation for the support of distribution and location transparency [22] of actors, the queuing layer which allows for different implementations of the message queues, and

finally, the processing layer which implements the actual execution of the messages. Each layer allows for further customization by means of plugins. The implementation is available at `https://github.com/CrispOSS/abs-api`.

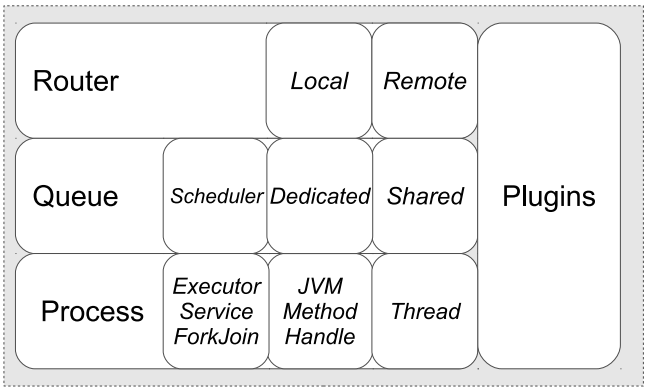

Fig. 1: Architecture of Actor API in Java 8

We discuss the architecture from bottom layer to top. The implementation of actor API preserves a faithful mapping of message processing in ABS modeling language. An actor is an active object in the sense that it controls how the next message is executed and may release any resources to allow for cooperative scheduling. Thus, the implementation is required to optimally utilize JVM threads. Clearly, allocating a dedicated thread to each message or actor is not scalable. Therefore, actors need to *share* threads for message execution and yet be in full control of resources when required. The implementation fundamentally separates *invocation* from *execution*. An asynchronous message is a reference to a method invocation until it starts its execution. This allows to minimize the allocation of threads to the messages and facilitates sharing threads for executing messages. Java concurrent API [23] provides different ways to deploy this separation of invocation from execution. We take advantage of Java Method Handles [29] to encapsulate invocations. Further we utilize different forms of `ExecutorService` and `ForkJoinPool` to deploy concurrent invocations of messages in different actors.

In the next layer, the actor API allows for different implementations of a queue for an actor. A dedicated queue for each actor simplifies the process of queuing messages for execution but consumes more resources. However, a shared queue for a set of actors allows for memory and storage optimization. This latter approach of deployment, first, provides a way to utilize the computing power of multi-core; for instance, it allows to use work-stealing to maximize the usage of thread pools. Second, it enables application-level scheduling of messages. The different implementations cater for a variety of

plugins, like one that releases computation as long as there is no item in the queue and becomes active as soon as an item is placed into the queue; e.g. `java.util.concurrent.BlockingQueue`. Further, different plugins can be injected to allow for scheduling of messages extended with deadlines and priorities [28].

We discuss next the distribution of actors in this architecture. In the architecture presented in Figure 1, each layer can be *distributed* independently of another layer in a transparent way. Not only the routing layer can provide distribution, the queue layer of the architecture may also be remote to take advantage of cluster storage for actor messages. A remote routing layer can provide access to actors transparently through standard naming or addresses. We exploit the main properties of actor model [1,2] to distribute actors based on our implementation. From a distributed perspective, the following are the main requirements for distributing actors:

Reference Location Transparency Actors communicate to one another using references. In an actor model, there is no in-memory object reference; however, every actor reference denotes a location by means of which the actor is accessible. The reference location may be local to the calling actor or remote. The reference location is *physically* transparent for the calling actor.

Communication Transparency A message m from actor A to actor B may possibly lead to transferring m over a network such that B can process the message. Thus, an actor model that supports distribution must provide a layer of remote communication among its actors that is transparent, i.e., when actor A sends message m, the message is transparently transferred over the network to reach actor B. For instance, actors existing in an HTTP container that transparently allows such communication. Further, the API implementation is required to provide a mechanism for serialization of messages. By default, every object in JVM cannot be assumed to be an instance of `java.io.Serializable`. However, the API may enforce that any remote actor should have the required actor classes in its JVM during runtime which allows the use of the JVM's general object serialization [6] to send messages to remote actors and receive their responses. Additionally, we model asynchronous messages with lambda expressions for which Java 8 supports serialization by specification [7].

Actor Provisioning During a life time of an actor, it may need to create new actors. Creating actors in a local memory setting is straightforward. However, the local setting *does* have a capacity of number of actors it can hold. When an actor creates a new one, the new actor may actually be initialized in a remote resource. When the resource is not available, it should be first provisioned. However, this resource provisioning should be transparent to the actor and only the eventual result (the newly created actor) is visible.

[6] Java Object Serialization Specification: `http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`

[7] Serialized Lambdas: `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SerializedLambda.html`

We extend the ABS API to ABS Remote API[8] that provides the above properties for actors in a seamless way. A complete example of using the remote API has been developed[9]. Expanding our ping-pong example in this paper, Listing 17 and 18 present how a remote server of actors is created for the ping and pong actors. In the following listings, `java.util.Properties` is used provide input parameters of the actor server; namely, the address and the port that the actor server responds to.

Listing 17: Remote ping actor main

```
1  Properties p =  new Properties();
2  p.put("host", "localhost");
3  p.put("port", "7777");
4  ActorServer s = new ActorServer(p)
       ;
5  IPong pong =
6   s.newRemote("abs://pong@http://
        localhost:8888",
7    IPong.class);
8  Ping ping = new Ping(pong);
9  ping.send(
10   () -> ping.ping("")
11 );
```

Listing 18: Remote pong actor main

```
1  Properties p = new Properties();
2  p.put("host", "localhost");
3  p.put("port", "8888");
4  ActorServer s = new ActorServer(p)
       ;
5  Pong pong = new Pong();
```

In Listing 17, a remote reference to a pong actor is created that exposes the `IPong` interface. This interface is proxied [10] by the implementation to handle the remote communication with the actual pong actor in the other actor server. This mechanism hides the communication details from the ping actor and as such allows the ping actor to use the same API to send a message to the pong actor (without even knowing that the pong actor is actually remote). When an actor is initialized in a distributed setting it transparently identifies its actor server and registers with it. The above two listings are aligned with the similar main program presented in Listing 15 that presents the same in a local setting. The above two listings run in separate JVM instances and therefore do not share any objects. In each JVM instance, it is required that both interfaces `IPing` and `IPong` are visible to the classpath; however, the ping actor server only needs to see `Ping` class in its classpath and similarly the pong actor server only needs to see `Pong` class in its classpath.

[8] The implementation is available at `https://github.com/CrispOSS/abs-api-remote`.

[9] An example of ABS Remote API is available at `https://github.com/CrispOSS/abs-api-remote-sample`.

[10] Java Proxy: `http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html`

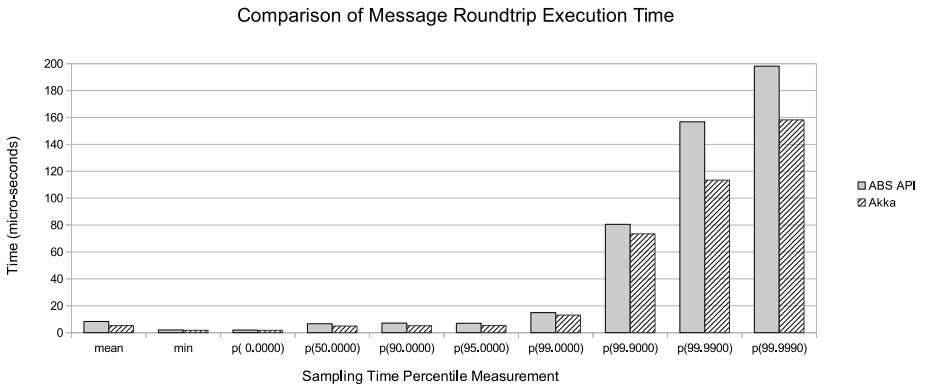

Fig. 2: Benchmark results of comparing sampling time of message round trips in ABS API and Akka. An example reading of above results is that the time shows for $p(90.0000)$ reads as "message round trips were completed under 10µs for 90% of the sent messages". The first two columns show the "minimum" and "mean" message round trip times in both implementations.

8 Experiments

In this section, we explain how a series of benchmarks were directed to evaluate the performance and functionality of actor API in Java 8. For this benchmark, we use a simple Java application that uses the "Ping-Pong" actor example discussed previously. An application consists of one instance of `Ping` actor and one instance of `Pong` actor. The application sends a `ping` message to the ping actor and waits for the result. The ping message depends on a `pong` message to the pong actor. When the result from the pong actor is ready, the ping actor completes the message; this completes a round trip of a message in the application. To be able to make comparison of how actor API in Java 8 performs, the example is also implemented using Akka [32] library. The same set of benchmarks are performed in isolation for both of the applications. To perform the benchmarks, we use JMH [30] that is a Java microbenchmarking harness developed by OpenJDK community and used to perform benchmarks for the Java language itself.

The benchmark is performed on the round trip of a message in the application. The benchmark starts with a warm-up phase followed by the running phase. The benchmark composes of a number of iterations in each phase and specific time period for each iteration specified for each phase. Every iteration of the benchmark triggers a new message in the application and waits for the result. The measurement used is *sampling time* of the round trip of a message. A specific number of samples are collected. Based on the samples in different phases, different *percentile* measurements are summarized. An example per-

centile measurement $p(99.9900) = 10\ \mu s$ is read as 99.9900% of messages in the benchmark took 10 micro-seconds to complete.

Each benchmark starts with 500 iterations of warm-up with each iteration for 1 micro-second. Each benchmark runs for 5000 iterations with each iteration for 50 micro-seconds. In each iteration, a maximum number of 50K samples are collected. Each benchmark is executed in an isolated JVM environment with Java 8 version b127. Each benchmark is executed on a hardware with 8 cores of CPU and a maximum memory of 8GB for JVM.

The results are presented in Figure 2. The performance difference observed in the measurements can be explained as follows. An actor in Akka is expected to expose a certain behavior as discussed in Section 3 (i.e. `onReceive`). This means that every message leads to an eventual invocation of this method inside actor. However, in case of an actor in Java 8, there is a need to make a look-up for the actual method to be executed with expected arguments. This means that for every method, although in the presence of caching, there is a need to find the proper method that is expected to be invoked. A constant overhead for the method look-up in order to adhere to the object-oriented principles is naturally to be expected. Thus, this is the minimal performance cost that the actor API in Java 8 pays to support programming to interfaces.

9 Conclusion

In this paper, we discussed an implementation of the actor-based ABS modeling language in Java 8 which supports the basic object-oriented mechanisms and principles of method look-up and programming to interfaces. In the full version of this paper we have developed an operational semantics of Java 8 features including lambda expressions and have proved formally the correctness of the embedding in terms of a bisimulation relation.

The underlying modeling language has an executable semantics and supports a variety of formal analysis techniques, including deadlock and schedulability analysis [10,19]. Further it supports a formal behavioral specification of interfaces [14], to be used as contracts.

We intend to expand this work in different ways. We aim to automatically *generate* ABS models from Java code which follows the ABS design methodology. Model extraction allows industry level applications be abstracted into models and analyzed for different goals such as deadlock analysis and concurrency optimization. This approach of model extraction we believe will greatly enhance industrial uptake of formal methods. We aim to further extend the implementation of API to support different features especially regarding distribution of actors especially in the queue layer, and scheduling of messages using application-level policies or real-time properties of a concurrent system. Furthermore, the current implementation of ABS API in a distributed setting allows for instantiation of remote actors. We intend to use the implementation to model ABS deployment components [21] and simulate a distributed environment.

References

1. G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, 1997.
2. Gul Agha. The Structure and Semantics of Actor Languages. In *Proc. the REX Workshop*, pages 1–59, 1990.
3. Joe Armstrong. Erlang. *Communications of ACM*, 53(9):68–75, 2010.
4. Michael Baker and Martin Thompson. *LMAX Disruptor*. LMAX Exchange. `http://github.com/LMAX-Exchange/disruptor`.
5. Po-Hao Chang and Gul Agha. Towards Context-Aware Web Applications. In *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 239–252, 2007.
6. Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks. In *Proc. Embedded net. sensor sys., SenSys 2005*, pages 302–302, 2005.
7. Ben Christensen. *RxJava: Reactive Functional Progamming in Java*. Netflix. `http://github.com/Netflix/RxJava/wiki`.
8. Martin Fowler. *LMAX Architecture*. Martin Fowler. `http://martinfowler.com/articles/lmax.html`.
9. Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93 – Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin Heidelberg, 1993.
10. Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *IFM*, pages 394–411, 2013.
11. Brian Goetz. *Lambda Expression Translation in Java 8*. Oracle. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.
12. Brian Goetz. *Lambda: A Peek Under The Hood*. Oracle, 2012. JAX London.
13. Brian Goetz. *JSR 335, Lambda Expressions for the Java Programming Language*. Oracle, March 2013. `http://jcp.org/en/jsr/detail?id=335`.
14. Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. Hats abstract behavioral specification: The architectural view. In *FMCO*, pages 109–132, 2011.
15. Philipp Haller. On the integration of the actor model in mainstream technologies: the Scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.
16. Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
17. Carl Hewitt. Procedural Embedding of knowledge in Planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
18. Carl Hewitt. What Is Commitment? Physical, Organizational, and Social (Revised). In *Proc. Coordination, Organizations, Institutions, and Norms in Agent Systems II*, LNCS Series, pages 293–307. Springer, 2007.
19. Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Log. Algebr. Program.*, 78(5):402–416, 2009.
20. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.

21. Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Formal Methods and Software Engineering*, pages 71–86. Springer, 2012.
22. Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proc. Principles and Practice of Prog. in Java (PPPJ'09)*, pages 11–20. ACM, 2009.
23. Doug Lea. *JSR 166: Concurrency Utilities*. Sun Microsystems, Inc. `http://jcp.org/en/jsr/detail?id=166`.
24. Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions in Embedded Computing Systems*, 8(4), 2009.
25. Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
26. Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51, Oct 1992.
27. Microsoft. *Reactive Extensions*. Microsoft. `https://rx.codeplex.com/`.
28. Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888. ACM, 2012.
29. John Rose. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform*. Oracle. `http://jcp.org/en/jsr/detail?id=292`.
30. Aleksey Shipilev. *JMH: Java Microbenchmark Harness*. Oracle. `http://openjdk.java.net/projects/code-tools/jmh/`.
31. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.
32. Typesafe. *Akka*. Typesafe. `http://akka.io/`.
33. Jose Valim. *Elixir*. Elixir. `http://elixir-lang.org/`.
34. Carlos A Varela, Gul Agha, Wei-Jen Wang, Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, and Abe Stephens. The SALSA Programming Language 1.1.2 Release Tutorial. *Dept. of Computer Science, RPI, Tech. Rep*, pages 07–12, 2007.
35. Wirfs-Brock, Rebecca J. and Johnson, Ralph E. Surveying Current Research in Object-oriented Design. *Commun. ACM*, 33(9):104–124, 1990.