Type-parameterized Actors and Their Supervision (v6.6.5)

Jiansen HE
University of Edinburgh
jiansen.he@ed.ac.uk

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Philip Trinder
University of Glasgow
P.W.Trinder@glasgow.ac.uk

Abstract

The robustness of distributed message passing applications can be improved by (i) employing failure recovery mechanisms such as the supervision principle, or (ii) using typed messages to prevent ill-typed communication. The former approach is originally developed in the Erlang OTP (Open Telecom Platform) library. The later approach has been well explored in systems including the join-calculus and the typed π-calculus. Attempts of combing those two approaches has been made in two directions: type checking existing Erlang programs and implementing a statically typed actor library that supports the supervision principle. Unfortunately, current statically typed actor libraries either use dynamically typed messages (e.g. Akka), or do not support supervision (e.g. Cloud Haskell).

Implementing a statically typed actor library that also supports the supervision principle raises three problems. Firstly, distributed resources are dynamically typed whereas local implementation is statically type checked and does not need to consider ill-typed messages. Therefore, a mixture of statical and dynamical type checking is required to make sure that distributed resources are well-typed. Secondly, preventing unexpected messages to an actor requires that actor having different types when it communicates with others. Thirdly, an elegant notion of sub-typing is required for the supervision purpose so that a supervisor can interact with child actors of different types, even in systems where sub-typing is not directly supported.

This paper introduces the typed Akka library, TAkka, which resolves above problems. Although TAkka actor extends Akka actor and use the Scala `Manifest` class to serialize type information, we believe that similar improvements can be made to actor libraries in other languages.

We evaluate the TAkka library by re-implementing examples built from small and medium sized Erlang and Akka libraries. Results show that Akka programs can be gradually upgraded to TAkka equivalents with minimal runtime and code size overheads. TAkka programs have similar scalability to their Akka equivalents. Finally, we port the Chaos Monkey library for assessing application reliability and design a Supervision View library for dynamically capturing the structure of supervision trees.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Design, Languages, Reliability

Keywords actor, type, supervision tree, name server

1. Introduction

The Erlang/OTP (Open Telecom Platform) library [Ericsson AB. 2012a] was released in 1996 for writing Erlang code using the Actor model [Hewitt et al. 1973] together with five OTP design principles derived from ten years' experience of Erlang programming. The OTP Design principles, especially the supervision principle, made it easier to build reliable distributed applications [Armstrong 2007].

The notions of actors and supervision trees have been ported to statically typed languages including Scala and Haskell. Scala actor libraries including Scala Actors [Haller and Odersky 2006, 2007] and Akka [Typesafe Inc. (a) 2012; Typesafe Inc. (b) 2012] use dynamically typed messages even though Scala is a statically typed language. Cloud Haskell [Epstein et al. 2011], a recent actor library, supports both dynamically and statically typed messages, but does not support supervision. Can actors in supervision trees be statically typed?

The key claim in this paper is that actors in supervision trees can be typed by parameterizing the actor class with the type of messages it expects to receive. Type-parameterized actors benefit both users and developers of actor-based services. For users, sending ill-

typed messages is prevented at compile time. Because messages are usually transmitted asynchronously, it may be otherwise difficult to trace the source of errors at runtime, especially in distributed environments. For service developers, since unexpected messages are eliminated from the system, they can focus on the logic of the services rather than worrying about incoming messages of unexpected types. Implementing type-parameterized actors in a statically-typed language; however, requires solving following three problems.

1. A typed name server is required to retrieve actor references of specific types. A distributed system usually requires a name server that maps names of services to processes that implement that service. If processes are dynamically typed, this is usually implemented as a map from names to processes. Can this be adapted to cases where processes are statically typed?
2. Supervisor actors must interact with child actors of different types. Actors are structured in supervision trees to improve system reliability. Each actor in a supervision tree needs to handle messages within its specific interests and also messages from its supervisor. Is it practical to define a supervisor that communicates with children of different type parameters?
3. Actors which receive messages from distinct parties may suffer from the type pollution problem, in which case a party imports too much type information about an actor and can send the actor messages not expected from it. Systems built on layered architecture or the MVC model are often victims of the type pollution problem. As an actor receives messages from distinct parties using its sole channel, its type parameter is the union type of all expected message types. Therefore, unexpected messages can be sent to actors which naively publishes its type parameter or permits dynamically typed messages. Can a type-parameterised actor have different types of communication interface, i.e. typed actor reference, when published to parties.

Continuing a line of work on merging types with actor programming by Haller and Odersky [Haller and Odersky 2006, 2007] and Akka developers [Typesafe Inc. (b) 2012], along with the work on system reliability test by Netflix, Inc. [Netflix, Inc. 2013] and Luna [Luna 2013], this paper makes following contributions.

- It presents the design and implementation of a novel typed name server that maps typed names to values, for example actor references, of the corresponding type. The typed name server (Section 3.2) mixes static and dynamic type checking so that type errors are detected at the earliest opportunity. The implementation requires runtime support for type reflection and a notion of first class type descriptors. In Scala, the `Manifest` class provides such facility.
- It describes the design of the TAkka library. Sections 4.1 to 4.3 illustrate how type parameters are added to actor related classes to improve type safety. By separating the handler for system messages from the handler for user defined messages, Sections 4.5 and 4.6 show that type-parameterized actors can form supervision trees in the same manner as untyped actors. Section 4.7 compares the design of TAkka with alternatives adopted by other actor libraries.
- It shows that Akka programs can gradually migrate to their TAkka equivalents. Section 5 explains strategies of gradually upgrading Akka programs to their TAkka equivalents.
- It gives a straightforward solution to the type pollution problem (Section 6.1). We present a simple API to cast the type of an actor reference to its super type. The semantics of the API is easy to understand and does not require deep understanding of underlying concepts such as inheritance, polymorphism, and contravariant types.
- It gives a comprehensive evaluation of the TAkka library. Results in Section 6.2 confirm that using type parameterized actors sacrifice neither expressiveness nor correctness. Efficiency and scalability test in Section 6.3 shows that TAkka applications have little overhead at the initialization stage but have almost identical run-time performance and scalability compared to their Akka equivalents. In addition, two helper libraries (Section 6.4) are shipped with the TAkka library. The Chaos Monkey library are ported to test applications against intensive adverse conditions. The novel Supervision View library is designed and implemented to dynamically monitor changes of supervision tree.

2. Actor Programming

2.1 Actor Model and OTP Design Principles

The Actor model defined by Hewitt et al. [Hewitt et al. 1973] treats actors as primitive computational components. Actors collaborate by sending asynchronous messages to each other. An actor independently determines its reaction to messages it receives.

The Actor model is adopted by the Erlang programming language, whose developers later summarized 5 OTP design principles to improve the reliability of Erlang applications [Ericsson AB. 2012b]. We notice that the Behaviour principle, the Application principle, and the Release principle coincide with 3 Java and Scala Programming practices listed in Table 1. The Release

OTP Design Principle	JAVA/Scala Programming
Behaviour	defining an abstract class
Application	defining an abstract class that has two abstract methods: start and stop
Release	packaging related application classes
Release Handling	hot swapping support on key modules is required
Supervision	no direct correspondence

Table 1. Using OTP Design Principles in JAVA/Scala Programming

Handling principle requires runtime support for hot swapping. In a platform where hot swapping is not supported in general, e.g. Java Virtual Machine (JVM), hot swapping a particular component can be simulated by updating the reference to that component. For example, Section 4.4 explains how to hot swap the behaviour of an actor in TAkka, which runs on the JVM. The Supervision principle, which is the central topic of this paper, has no direct correspondence in native JVM based systems.

2.2 Akka Actor

An Akka Actor has four essential components as shown in Figure 1: (i) a receive function that defines its reaction to incoming messages, (ii) an actor reference pointing to itself, (iii) the actor context representing the outside world of the actor, and (iv) the supervisor strategy for its children.

```
1 package akka.actor
2 trait Actor{
3   def receive:Any=>Unit
4   val self:ActorRef
5   val context:ActorContext
6   var supervisorStrategy: SupervisorStrategy
7 }
```

Figure 1. Akka Actor API

Figure 2 shows an example actor in Akka. The `receive` function of the Akka actor has type `Any⇒Unit` but the defined actor, `ServerActor`, is only intended to process strings. At Line 16, a `Props`, an abstraction of actor creation, is initialized and passed to an actor system, which creates an actor with name `server` and returns a reference pointing to that actor. Another way to obtain an actor is using the `actorFor` method as shown in line 24. We then use actor references to send the actor string messages and integer messages. String

```
1 class ServerActor extends Actor {
2   def receive = {
3     case m:String => println("received message:
          "+m)
4   }
5 }
6
7 class MessageHandler(system: ActorSystem) extends
      Actor {
8   def receive = {
9     case akka.actor.UnhandledMessage(message,
          sender, recipient) =>
10      println("unhandled message:"+message);
11  }
12 }
13
14 object ServerTest extends App {
15   val system = ActorSystem("ServerTest")
16   val server = system.actorOf(Props[ServerActor],
       "server")
17
18   val handler = system.actorOf(Props(new
       MessageHandler(system)))
19   system.eventStream.subscribe(handler,
20                   classOf[akka.actor.UnhandledMessage]);
21   server ! "Hello World"
22   server ! 3
23
24   val serverRef =
       system.actorFor("akka://ServerTest/user/server")
25   serverRef ! "Hello World"
26   serverRef ! 3
27 }
28
29
30 /*
31 Terminal output:
32 received message: Hello World
33 unhandled message:3
34 received message: Hello World
35 unhandled message:3
36 */
```

Figure 2. A String Processor in Akka

messages are processed in the way defined by the receive function.

Undefined messages are treated differently in different actor libraries. In Erlang, an actor keeps undefined messages in its mailbox, attempts to process the message again when a new message handler is in use. In versions prior to 2.0, an Akka actor raises an exception when it processes an undefined message. In recent Akka versions, an undefined message is discarded by the actor and an `UnhandledMessage` event is pushed to the event stream of the actor system. The event stream may be subscribed by other actors who are interested

in particular event messages. To handle the unexpected integer message in the above short example, an event handler is defined and created with 6 lines of code.

2.3 Supervision

Reliable Erlang applications typically adopt the Supervision Principle [Ericsson AB. 2012b], which suggests that actors should be organized in a tree structure so that any failed actor can be properly restarted by its supervisor. Nevertheless, adopting the Supervision principle is optional in Erlang.

The Akka library makes supervision obligatory by restricting the way of creating actors. Actors can only be initialized by using the `actorOf` method provided by `ActorSystem` or `ActorContext`. Each actor system provides a guardian actor for all user-created actors. Calling the `actorOf` method of an actor system creates an actor supervised by the guardian actor. Calling the `actorOf` method of an actor context creates a child actor supervised by that actor. Therefore, all user-created actors in an actor system, together with the guardian actor of that actor system, form a tree structure. Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance.

Each actor in Akka is associated with an actor path. The string representation of the actor path of a guardian actor has format *akka://mysystem@IP:port/user*, where *mysystem* is the name of the actor system, *IP* and *port* are the IP address and the port number which the actor system listens to, and *user* is the name of the guardian actor. The actor path of a child actor is actor path of its supervisor appended by the name of the child actor, either a user specified name or a system generated name.

Figure 3 defines a simple calculator which supports multiplication and division. The simple calculator does not consider the problematic case of dividing a number by 0, where an `ArithmeticException` will be raised. We then define a safe calculator as the supervisor of the simple calculator. The safe calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` is raised. The supervisor strategy of the safe calculator also specifies the maximum number of failures its child may have within a given time range. If the child fails more frequently than the allowed frequency, the safe calculator will be stopped, and its failure will be reported to its supervisor, the system guardian actor in this example. The terminal output shows that the simple calculator is restarted before the third message and the fifth message are delivered. The last message is not processed because both calculators are terminated since the simple calculator fails more frequently than allowed.

```
case class Multiplication(m:Int, n:Int)
case class Division(m:Int, n:Int)

class Calculator extends Actor {
  def receive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}

class SafeCalculator extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 2,
        withinTimeRange = 1 minute) {
      case _: ArithmeticException =>
        println("ArithmeticException Raised to:
            "+self)
        Restart
    }
  val child:ActorRef =
      context.actorOf(Props[Calculator], "child")
  def receive = {
    case m => child ! m
  }
}

  val system = ActorSystem("MySystem")
  val actorRef:ActorRef =
      system.actorOf(Props[SafeCalculator],
"safecalculator")

  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 2)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 3)

/*
Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
     Actor[akka://MySystem/user/safecalculator]

10 / 5 = 2
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
     Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero

3 * 2 = 6
ArithmeticException Raised to:
     Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
*/
```

Figure 3. Supervised Calculator

3. Mixing Static and Dynamic Type Checking

A key advantages of static typing is that it detects some type errors at an early stage, i.e. at compile time. The TAkka library is designed to detect type errors as early as possible. Nevertheless, not all type errors can be statically detected, and some dynamic type checks are required. To address this issue, a notion of run-time type descriptor is required.

This section summarizes the type reflection mechanism in Scala and explains how it benefits the implementation of our typed name server. Our typed name server can be straightforwardly ported to other platforms that support type reflection.

3.1 Scala Type Descriptors

Scala 2.8 defines a `Manifest` class [1] whose instance is a first class type descriptor used at runtime. With the help of the `Manifest` class, users can record the type information, including generic types, which may be erased by the JAVA compiler.

In the Scala interactive session below, we obtain a Manifest value at Line 5 and test a subtype relationship at Line 8. To define a method that obtains type information of a generic type, Scala requires a type tag as an implicit argument to the method. To simplify the API, Scala further provides a form of syntactic sugar called context bounds. We define a method using context bounds at Line 11, which is compiled to the version using implicit arguments as shown at Line 12.

```
1 scala> class Sup; class Sub extends Sup
2 defined class Sup
3 defined class Sub
4
5 scala> manifest[Sub]
6 res0: Manifest[Sub] = Sub
7
8 scala> manifest[Sub] <:< manifest[Sup]
9 res1: Boolean = true
10
11 scala> def getType[T:Manifest] = {manifest[T]}
12 getType: [T](implicit evidence$1:
      Manifest[T])Manifest[T]
13
14 scala> getType[Sub => Sup => Int]
15 res2: Manifest[Sub => (Sup => Int)] =
      scala.Function1[Sub, scala.Function1[Sup,
      Int]]
```

[1] Scala 2.10 introduces the Type class and the TypeTag class to replace Manifest. At the time of this writing, TypeTag is not serializable as it should be due to bug SI-5919.

3.2 Typed Name Server

In distributed systems, a name server maps each registered name, usually a unique string, to a dynamically typed value, and provides a function to look up a value for a given name. A name can be encoded as a `Symbol` in Scala so that names which represent the same string have the same value. As a value retrieved from a name server is *dynamically typed*, it needs to be checked against and be cast to the expected type at the client side before using it.

To overcome the limitations of the untyped name server, we design and implement a typed name server which maps each registered typed name to a value of the corresponding type, and allows to look up a value by giving a typed name.

A typed name, `TSymbol`, is a name shipped with a type descriptor. A typed value, `TValue`, is a value shipped with a type descriptor, which describes a super type of the most precise type of that value. In Scala, `TSymbol` and `TValue` can be simply defined as in Figure 4:

```
1 case class TSymbol[-T:Manifest](val s:Symbol) {
2    private [takka] val t:Manifest[_] = manifest[T]
3    override def hashCode():Int = s.hashCode()
4 }
5
6 case class TValue[T:Manifest](val value:T){
7  private [takka] val t:Manifest[_] = manifest[T]
8 }
```

Figure 4. TSymbol and TValue

`TSymbol` is declared as a *case class* in Scala so that it can be used as a data constructor and for pattern matching. In addition, the type descriptor, `t`, is constructed automatically and is private to the `takka` package so that only the library developer can access it as a field of `TSymbol`. `TValue` is declared as a *case class* for the same reason.

With the help of `TSymbol`, `TValue`, and a hashmap, a typed name server provides the following three operations:

- `set[T:Manifest](name:TSymbol[T], value:T):Boolean`

  The `set` operation registers a typed name with a value of corresponding type and returns true if the symbol representation of *name* has not been registered; otherwise the typed name server discards the request and returns false.

- `unset[T](name:TSymbol[T]):Boolean`

  The `unset` operation cancels the entry *name* and returns true if (i) its symbol representation is registered and (ii) the type `T` is a supertype of the registered type; otherwise the operation returns false.

- `get[T] (name: TSymbol[T]): Option[T]`

  The `get` operation returns Some(v:T), where `v` is the value associated with `name`, if (i) *name* is associated with a value and (ii) `T` is a supertype of the registered type; otherwise the operation returns None.

Notice that `unset` and `get` operations succeed as long as the associated type of the input name is the supertype of the associated type of the registered name. To permit polymorphism, the `hashcode` method of `TSymbol` defined in Figure 4 does not take type values into account. Equivalence comparison on `TSymbol` instances, however, should consider the type. Although the notion of `TValue` does not appear in the API, it is required for an efficient library implementation because the type information in `TSymbol` is ignored in the hashmap. Overriding the hash function of `TSymbol` also prevents the case where users accidentally register two typed names with the same symbol but different types, in which case if one type is a supertype of the other, the return value of `get` can be non-deterministic. Last but not least, when an operation fails, the name server returns `false` or `None` rather than raising an exception so that it is always available.

In general, dynamic type checking can be carried out in two ways. The first method is to check whether the most precise type of a value conforms to the structure of a data type. Examples of this method include dynamically typed languages and the `instanceof` method in JAVA and other languages. The second method is to compare two type descriptors at run time. The implementation of our typed name server employs the second method because it detects type errors which may otherwise be left out. Our implementation requires the runtime type reification feature provided by Scala. In a system that does not have such a feature, implementing typed name servers is more difficult.

4. TAkka Library Design

This section presents the design of the TAkka library. We outline how we add types to actors and how to construct supervision trees of typed actors in TAkka. This section concludes with a brief discussion about design alternatives used by other actor libraries.

4.1 Type-parameterized Actor

A TAkka actor has type `TypedActor[M]`. It inherits the Akka `Actor` trait to minimize implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor APIs. Instead, we encourage programmers to use the typed fields given in Figure 5. Unlike other actor libraries, every TAkka actor class takes a type parameter `M` which specifies the type of messages it expects to receive. The same type parameter is used as the input type of the receive function, the type parameter of actor context and the type parameter of the actor self reference.

```
1 package takka.actor
2 abstract class TypedActor[M:Manifest] extends
      akka.actor.Actor {
3   def typedReceive:M=>Unit
4   val typedSelf:ActorRef[M]
5   val typedContext:ActorContext[M]
6   var supervisorStrategy: SupervisorStrategy
7 }
```

Figure 5. TAkka Actor API

The two immutable fields of `TypedActor`: `typedContext` and `typedSelf`, will be initialized automatically when the actor is created. Library users may override the default supervisor strategy in the way explained in Section 4.5. The implementation of the `typedReceive` method, on the other hand, is always provided by users.

The limitation of using inheritance to implement TAkka actors is that Akka features are still available to library users. Unfortunately, this limitation cannot be overcome by using delegation because, as we have seen in Figure 2, a child actor is created by calling the `actorOf` method from its supervisor's actor context, which is a private API of the supervisor. `TypedActor` is the only TAkka class that is implemented using inheritance. Other TAkka classes are either implemented by delegating tasks to Akka counterparts or rewritten in TAkka. We believe that re-implementing the TAkka Actor library requires a similar amount of work for implementing the Akka Actor library.

4.2 Actor Reference

A reference to an actor of type `TypedActor[M]` has type `ActorRef[M]`. An actor reference provides a `!` method, through which users can send a message to the referenced actor. Sending an actor a message whose type is not the expected type will raise an error at compile time. By using type-parameterized actor references, the receiver does not need to worry about unexpected messages, while senders can be sure that messages will be understood and processed, as long as the message is delivered.

An actor usually can react to a finite set of different message patterns whereas our notion of actor reference only takes one type parameter. In a type system that supports untagged union types, no special extension is required. In a type system which supports polymorphism, `ActorRef` should be contravariant on its type argument `M`, denoted as `ActorRef[-M]`. Consider rewriting the simple calculator defined in Section 2.3 using TAkka, it is clear that `ActorRef` is con-

travariant because `ActorRef[Operation]` is a subtype of `ActorRef[Division]` though `Division` is a subtype of `Operation`. contravariance is crucial to avoid the type pollution problem described at Section 6.1.

```
1 abstract class ActorRef[-M](implicit
      mt:Manifest[M]) {
2   def !(message: M):Unit
3   def publishAs[SubM<:M](implicit
        smt:Manifest[SubM]):ActorRef[SubM]
4 }
```

Figure 6. Actor Reference

For ease of use, TAkka provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method has three advantages. Firstly, explicit type conversion using `publishAs` is always type safe because the type of the result is a supertype of the original actor reference. Secondly, the semantics of `publishAs` does not require a deep understanding of underlying concepts like contravariance and inheritance. Thirdly, with the `publishAs` method, users can give a supertype of an actor reference on demand, without defining new types and recompiling affected classes in the type hierarchy.

Figure 2 defines the same string processing actor given in Section 2.2. The `typedReceive` method now has type `String⇒Unit`, which is the same as intended. In this example, the types of `typedReceive` and `m` may be omitted because they can be inferred by the compiler. Unlike the Akka example, sending an integer to `server` is rejected by the compiler. Although the type error introduced on line 19 cannot be statically detected, it is captured by the run-time as soon as the `actorFor` method is called. In the TAkka version, there is no need to define a handler for unexpected messages.

4.3 Props and Actor Context

The type `Props` denotes the properties of an actor. A Props of type `Props[M]` is used when creating an actor of type `TypedActor[M]`. Say `myActor` is of type `MyActor`, which is a subtype of `TypedActor[M]`, a Prop of type `Prop[M]` can be created by one of the APIs in Figure 8:

Contrary to an actor reference, which is the interface for receiving messages, an actor context describes the actor's view of the outside world. Because each actor is an independent computational primitive, an actor context is private to the corresponding actor. By using APIs in Figure 9, an actor can (i) retrieve an actor reference corresponding to a given actor path using the `actorFor` method, (ii) create a child actor with a system-generated or user-specified name using one of the `actorOf` methods, (iii) set a timeout denoting the time within which a new message must be received

```
1 class ServerActor extends TypedActor[String] {
2   def typedReceive = {
3     case m:String => println("received message:
          "+m)
4   }
5 }
6
7 object ServerTest extends App {
8   val system = ActorSystem("ServerTest")
9   val server = system.actorOf(Props[String,
        ServerActor], "server")
10
11  server ! "Hello World"
12 //  server ! 3
13 // compile error: type mismatch; found : Int(3)
14 //   required: String
15
16  val serverString = system.actorFor[String]
17                  ("akka://ServerTest/user/server")
18  serverString ! "Hello World"
19  val serverInt = system.actorFor[Int]
20                ("akka://ServerTest/user/server")
21  serverInt ! 3
22 }
23
24 /*
25 Terminal output:
26 received message: Hello World
27 received message: Hello World
28 Exception in thread "main" java.lang.Exception:
29  ActorRef[akka://ServerTest/user/server] does not
30  exist or does not have type ActorRef[Int] at
31  takka.actor.ActorSystem.actorFor(ActorSystem.scala:223)
32  ...
33 */
```

Figure 7. A String Processor in TAkka

```
1 val props:Props[M] = Props[M, MyActor]
2 val props:Props[M] = Props[M](new MyActor)
3 val props:Props[M] = Props[M](myActor.getClass)
```

Figure 8. Actor Props

using the `setReceiveTimeout` method, and (iv) update its behaviours using the `become` method. Comparing corresponding Akka APIs, our methods take an additional type parameter whose meaning will be explained below.

The two `actorOf` methods are used to create a type-parameterized actor supervised by the current actor. Each actor created is assigned to a typed actor path, an Akka actor path together with a `Manifest` of the message type. Each actor system contains a typed name server. When an actor is created inside an actor system,

```
abstract class ActorContext[M:Manifest] {
  def actorOf [Msg] (props: Props[Msg])(implicit
      mt: Manifest[Msg]): ActorRef[Msg]
  def actorOf [Msg] (props: Props[Msg], name:
      String)(implicit mt: Manifest[Msg]):
      ActorRef[Msg]
  def actorFor [Msg] (actorPath: String)
      (implicit mt: Manifest[Msg]): ActorRef[Msg]
  def setReceiveTimeout(timeout: Duration): Unit

  def become[SupM >: M](
     newTypedReceive: SupM => Unit,
     newSystemMessageHandler:
                      SystemMessage => Unit,
     newSupervisorStrategy:SupervisorStrategy
  )(implicit smt:Manifest[SupM]):ActorRef[SupM]
}
```

Figure 9. Actor Context

a mapping from its typed actor path to its typed actor reference is registered to the typed name server. The `actorFor` method of `ActorContext` and `ActorSystem` fetches typed actor reference from the typed name server.

4.4 Backward Compatible Hot Swapping

Hot swapping is a desired feature of distributed systems, whose components are typically developed separately. Unfortunately, hot swapping is not supported by the JVM, the platform on which the TAkka library runs. To support hot swapping on an actor's receive function, system message handler, and supervisor strategy, those three behaviour methods are maintained as object references.

The `become` method enables hot swapping on the behaviour of an actor. The `become` method in TAkka is different from behaviour upgrades in Akka in two aspects. Firstly, the supervisor strategy can be updated. In Akka, the supervisor strategy is an immutable value of an actor. We believe the supervisor strategy is an important behaviour of an actor and it should be as swappable as message handlers. Secondly, hot swapping in TAkka must be backward compatible. In other words, an actor must evolve to a version that is able to handle the original message patterns. The above decision is made so that a service published to users will not be unavailable later.

The `become` method is implemented as in Figure 10. The static type `M` should be interpreted as the least general type of messages addressed by the actor initialized from `TypedActor[M]`. The type value of `SupM` will only be known when the `become` method is invoked. When a series of `become` invocations are made at run time, the order of those invocations may be non-deterministic.

```
abstract class ActorContext[M:Manifest] {
  implicit private var mt:Manifest[M] = manifest[M]

  def become[SupM >: M](
     newTypedReceive: SupM => Unit,
     newSystemMessageHandler:
             SystemMessage => Unit
     newSupervisorStrategy:SupervisorStrategy
  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM]
       = {
    val smt = manifest[SupM]
    if (!(mt <:< smt))
      throw BehaviorUpdateException(smt, mt)

    this.mt = smt
    this.systemMessageHandler =
        newSystemMessageHandler
    this.supervisorStrategy = newSupervisorStrategy
  }
}
```

Figure 10. Hot Swapping in TAkka

Therefore, performing dynamic type checking is required to guarantee backward compatibility. Nevertheless, static type checking prevents some invalid `become` invocations at compile time.

4.5 Supervisor Strategies

The Akka library implements two of the three supervisor strategies in OTP: `OneForOne` and `AllForOne`. If a supervisor adopts the `OneForOne` strategy, a child will be restarted when it fails. If a supervisor adopts the `AllForOne` supervisor strategy, all children will be restarted when any of them fails. The third OTP supervisor strategy, `RestForOne`, restarts children in a user-specified order, and hence is not supported by Akka as it does not specify an order of initialization for children. Simulating the `RestForOne` supervisor strategy in Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. None of the Erlang examples in Section 5 uses the `RestForOne` strategy. It is not clear whether the lack of the `RestForOne` strategy will result in difficulties when rewriting Erlang applications in Akka and TAkka.

Figure 11 gives APIs of supervisor strategies in Akka. As in OTP, for each supervisor strategy, users can specify the maximum number of restarts of any child within a period. The default supervisor strategy in Akka is `OneForOne` that permits unlimited restarts. `Directive` is an enumerated type with the following values: the `Escalate` action which throws the exception to the supervisor of the supervisor, the `Restart` action which replaces the failed child with a new one, the

`Resume` action which asks the child to process the message again, and the `Stop` action which terminates the failed actor permanently.

None of the supervisor strategies in Figure 11 requires a type-parameterized classes during construction. Therefore, from the perspective of API design, both supervisor strategies are constructed in TAkka in the same way as in Akka.

```
1 abstract class SupervisorStrategy
2 case class OneForOne(restart:Int,
    time:Duration)(decider: Throwable =>
    Directive) extends SupervisorStrategy
3 case class OneForAll(restart:Int,
    time:Duration)(decider: Throwable =>
    Directive) extends SupervisorStrategy
```

Figure 11. Supervisor Strategies

4.6 Handling System Messages

Actors communicate with each other by sending messages. To maintain a supervision tree, a special category of messages should be handled by all actors. We define a trait [2] `SystemMessage` to be the supertype of all messages for system maintenance purposes. The five Akka system messages retained in TAkka are given as follows:

- `ChildTerminated(child: ActorRef[M])`

  A message sent from a child actor to its supervisor before it terminates.

- `Kill`

  An actor that receives this message will send an `ActorKilledException` to its supervisor.

- `PoisonPill`

  An actor that receives this message will be permanently terminated. The supervisor cannot restart the killed actor.

- `Restart`

  A message sent from a supervisor to its terminated child asking the child to restart.

- `ReceiveTimeout`

  A message sent from an actor to itself when it has not received a message after a timeout.

The next question is whether a system message should be handled by the library or by users. In Erlang and early versions of Akka, all system messages can be explicitly handled by users in the `receive` block. In recent Akka versions, some system messages are handled in the library implementation and are not accessible by library users.

As there are only two kinds of supervisor strategies to consider, both of which have clearly defined operational behaviours, all messages related to the liveness of actors are handled in the TAkka library. Library users may indirectly affect the system message handler via specifying the supervisor strategies. In contrast, messages related to the behaviour of an actor, e.g. `ReceiveTimeout`, are better handled by application developers. In TAkka, `ReceiveTimeout` is the only system message that can be explicitly handled by users. Nevertheless, we keep the `SystemMessage` trait in the library so that new system messages can be included in the future when required.

A key design decision in TAkka is to separate handlers for the system messages and user-defined messages. The above decision has two benefits. Firstly, the type parameter of actor-related classes only need to denote the type of user defined messages rather than the untagged union of user defined messages and the system messages. Therefore, the TAkka design applies to systems that do not support untagged union type. Secondly, since system messages can be handled by the default handler, which applies to most applications, users can focus on the logic of handling user defined messages.

4.7 Design Alternatives

Akka Typed Actor In the Akka library, there is a special class called `TypedActor`, which contains an internal actor and can be supervised. A service of `TypedActor` is invoked by method invocation instead of message exchanging. Code in Figure 12 demonstrates how to define a simple string processor using Akka typed actor. The Akka `TypedActor` prevent some type errors but have two limitations. Firstly, `TypedActor` does not permit hot swapping on its behaviours. Secondly, avoiding type pollution by using Akka typed actors is as awkward as using a plain object-oriented model, where supertypes need to be introduced. In Scala and Java, introducing a supertype in a type hierarchy requires modification to all affected classes.

Actors with or without Mutable States The actor model formalized by Hewitt et al. [Hewitt et al. 1973] does not specify its implementation strategy. In Erlang, a functional programming language, actors do not have mutable states. In Scala, an object-oriented programming language, actors may have mutable states. The TAkka library is built on top of Akka and implemented in Scala. As a result, TAkka does not prevent users from defining actors with mutable states. Nevertheless, the authors of this paper encourage the use of actors in a functional style, for example encoding the `sender` of a

[2] A trait in Scala is similar to a JAVA abstract class, but trait permits multiple inheritance.

```
1 trait MyTypedActor{
2   def processString(m:String)
3 }
4 class MyTypedActorImpl(val name:String) extends
      MyTypedActor{
5   def this() = this("default")
6
7   def processString(m:String) {
8     println("received message: "+m)
9   }
10 }
11 object FirstTypedActorTest extends App {
12   val system = ActorSystem("MySystem")
13   val myTypedActor:MyTypedActor =
14     TypedActor(system).typedActorOf(
15     TypedProps[MyTypedActorImpl]())
16   myTypedActor.processString("Hello World")
17 }
18
19 /*
20 Terminal output:
21 received message: Hello World
22  */
```

Figure 12. Akka TypedActor Example

synchronous message as part of the incoming message rather than a state of an actor, because it is difficult to synchronize mutable states of replicated actors in a cluster environment.

In a cluster, resources are replicated at different locations to provide fault-tolerant services. The CAP theorem [Gilbert and Lynch 2002] shows that it is impossible to achieve consistency, availability, and partition tolerance in a distributed system simultaneously. For actors that use mutable state, system providers must either sacrifice availability or partition tolerance, or modify the consistency model. For example, Akka actors have mutable state and Akka cluster developers spend great effort to implement an eventual consistency model [Kuhn et al. 2012]. In contrast, stateless services, e.g. RESTful web services, are more likely to achieve a good scalability and availability.

Bi-linked Actors In addition to one-way linking in the supervision tree, Erlang and Akka provide a mechanism to define a two-way linkage between actors. Bi-linked actors are each aware of the liveness of the other. We believe that bi-linked actors are redundant in a system where supervision is obligatory. Notice that, if the computation of an actor relies on the liveness of another actor, those two actors should be organized in the same supervision tree.

5. Evolution, Not Revolution

Akka systems can be smoothly migrated to TAkka systems. In other words, existing systems can evolve to introduce more types, rather than requiring a revolution where all actors and interactions must be typed.

The above property is analogous to adding generics to Java programs. Java generics are carefully designed so that programs without generic types can be partially replaced by an equivalent generic version (evolution), rather than requiring generic types everywhere (revolution) [Naftalin and Wadler 2006].

In previous sections, we have seen how to use Akka actors in an Akka system (Figure 2) and how to use TAkka actors in a TAkka system (Figure 7). In the following, we will explain how to use TAkka actors in an Akka system and how to use an Akka actor in a TAkka system.

5.1 TAkka actor in Akka system

It is often the case that an actor-based library is implemented by one organization but used in a client application implemented by another organization. If a developer decides to upgrade the library implementation using TAkka actors, for example, by upgrading the Socko Web Server [Imtarnasan and Bolton 2012], the Gatling [Excilys Group 2012] stress testing tool, or the core library of the Play framework [Typesafe Inc. (c) 2013] as we have done in Section 6.2, how the upgrade will affect client code, especially legacy applications built using the Akka library? TAkka actors and actor references are implemented using inheritance and delegation respectively so that no changes are required for legacy applications.

TAkka actors inherits Akka actors. In Figure 13, the actor implementation is upgraded to the TAkka version as in Figure 7. The client code, line 15 through line 25, is the same as the old Akka version given in Figure 2. That is, no changes are required for the client application.

TAkka actor reference delegates the task of message sending to an Akka actor reference, its `untypedRef` field. In line 31 in Figure 2, we get an untyped actor reference from `typedserver` and use the untyped actor reference in code where an Akka actor reference is expected. Because an untyped actor reference accepts messages of any type, messages of unexpected type may be sent to TAkka actors if an Akka actor reference is used. As a result, users who are interested in the `UnhandledMessage` event may subscribe to the event stream as in line 33.

5.2 Akka Actor in TAkka system

Sometimes, developers want to update the client code or the API before upgrading the actor implementation. For example, a developer may not have access to the

```
class TAkkaServerActor extends
    takka.actor.TypedActor[String] {
  def typedReceive = {
    case m:String => println("received message:
        "+m)
  }
}

class MessageHandler(system:
    akka.actor.ActorSystem) extends
    akka.actor.Actor {
  def receive = {
    case akka.actor.UnhandledMessage(message,
        sender, recipient) =>
     println("unhandled message:"+message);
  }
}

object TAkkaInAkka extends App {
  val akkasystem =
      akka.actor.ActorSystem("AkkaSystem")
  val akkaserver = akkasystem.actorOf(
    akka.actor.Props[TAkkaServerActor], "server")

  val handler = akkasystem.actorOf(
    akka.actor.Props(new
        MessageHandler(akkasystem)))

  akkasystem.eventStream.subscribe(handler,
     classOf[akka.actor.UnhandledMessage]);
  akkaserver ! "Hello Akka"
  akkaserver ! 3

  val takkasystem =
      takka.actor.ActorSystem("TAkkaSystem")
  val typedserver = takkasystem.actorOf(
     takka.actor.Props[String, ServerActor],
         "server")

  val untypedserver = takkaserver.untypedRef

  takkasystem.system.eventStream.subscribe(
    handler,classOf[akka.actor.UnhandledMessage]);

  untypedserver ! "Hello TAkka"
  untypedserver ! 4
}

/*
Terminal output:
received message: Hello Akka
unhandled message:3
received message: Hello TAkka
unhandled message:4
*/
```

Figure 13. TAkka actor in Akka application

```
class AkkaServerActor extends akka.actor.Actor {
  def receive = {
    case m:String => println("received message:
        "+m)
  }
}

object AkkaInTAkka extends App {
  val system = akka.actor.ActorSystem("AkkaSystem")
  val akkaserver = system.actorOf(
      akka.actor.Props[AkkaServerActor], "server")

  val takkaServer = new
      takka.actor.ActorRef[String]{
    val untypedRef = akkaserver
  }

  takkaServer ! "Hello World"
//  takkaServer ! 3
}

/*
Terminal output:
received message: Hello World
*/
```

Figure 14. Akka actor in TAkka application

actor code; or the library may be large, so the developer may want to upgrade the library gradually.

Users can initialize a TAkka actor reference by providing an Akka actor reference and a type parameter. In Figure 14, we re-use the Akka actor, initialize the actor in an Akka actor system, and obtain an Akka actor reference as in Figure 2. Then, we initialize a TAkka actor reference, `takkaServer`, which only accepts `String` messages.

6. Library Evaluation

This section presents the preliminary evaluation results of the TAkka library. We show that the Wadler's type pollution problem can be avoided in a straightforward way by using TAkka. We further assess the TAkka library by porting examples written in Erlang and Akka. Results show that TAkka detects type errors without causing obvious runtime and code-size overheads.

6.1 Wadler's Type Pollution Problem

Wadler's type pollution problem refers to the situation where a communication interface of a component publishes too much type information to another party and consequently that party can send the component a message not expected from it. Without due care, actor-based systems constructed using the layered architec-

ture or the MVC model can suffer from the type pollution problem.

One solution to the type pollution problem is using separate channels for distinct parties. Programming models that support this solution includes the join-calculus [Fournet and Gonthier 2000] and the typed π-calculus [Sangiorgi and Walker 2001].

TAkka solves the type pollution problem by using polymorphism. Take the code template in Figure 15 for example. Let `V2CMessage` and `M2CMessage` be the type of messages expected from the View and the Model respectively. Both `V2CMessage` and `M2CMessage` are subtypes of `ControllerMsg`, which is the least general type of messages expected by the controller. In the template code, the controller publishes itself as different types to the view actor and the model actor. Therefore, both the view and the model only know the communication interface between the controller and itself. The `ControllerMsg` is a sealed trait so that users cannot define a subtype of `ControllerMsg` outside the file and send the controller a message of unexpected type. Although type convention in line 25 and line 27 can be omitted, we explicitly use the `publishAs` to express our intention and let the compiler check the type. The code template is used to implement the Tik-Tak-Tok example in the TAkka code repository.

6.2 Expressiveness and Correctness

Table 2 lists the examples used for expressiveness and correctness. We selected examples from Erlang Quiviq [Arts et al. 2006] and open source Akka projects to ensure that the main requirements for actor programming are not unintentionally neglected. Examples from Erlang Quiviq are re-implemented using both Akka and TAkka. Examples from Akka projects are re-implemented using TAkka. Following the suggestion in [Hennessy and Patterson 2006], we assess the overall code modification and code size by calculating the geometric mean of all examples. The evaluation results in Table 3 show that when porting an Akka program to TAkka, about 7.4% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because TAkka code does not need to handle unexpected messages. On average, the total program size of Akka and TAkka applications are almost the same.

A type error is reported by the compiler when porting the Socko example [Imtarnasan and Bolton 2012] from its Akka implementation to equivalent TAkka implementation. SOCKO is a library for building event-driven web services. The SOCKO designer defines a `SockoEvent` class to be the supertype of all events. One subtype of `SockoEvent` is `HttpRequestEvent`, representing events generated when an HTTP request is received. The designer further implements subclasses

```
1  sealed trait ControllerMsg
2  class V2CMessage extends ControllerMsg
3  class M2CMessage extends ControllerMsg
4
5  trait C2VMessage
6  case class
       ViewSetController(controller:ActorRef[V2CMessage])
       extends
7  C2VMessage
8  trait C2MMessage
9  case class
       ModelSetController(controller:ActorRef[M2CMessage])
       extends
10 C2MMessage
11
12 class View extends TypedActor[C2VMessage] {
13   private var controller:ActorRef[V2CMessage]
14   // rest of implementation
15 }
16 Model extends TypedActor[C2MMessage] {
17   private var controller:ActorRef[M2CMessage]
18   // rest of implementation
19 }
20
21 class Controller(model:ActorRef[C2MMessage],
       view:ActorRef[C2VMessage]) extends
22 TypedActor[ControllerMessage] {
23   override def preStart() = {
24     model ! ModelSetController(
25             typedSelf.publishAs[M2CMessage])
26     view ! ViewSetController(
27           typedSelf.publishAs[V2CMessage])
28   }
29   // rest of implementation
30 }
```

Figure 15. Template for Model-View-Controller

of `Method`, whose `unapply` method intends to pattern match `SockoEvent` to `HttpRequestEvent`. The SOCKO designer made a type error in the method declaration so that the `unapply` method pattern matches `SockoEvent` to `SockoEvent`. The type error is not exposed in test examples because the those examples always passes instances of `HttpRequestEvent` to the `unapply` method and send the returned values to an actor that accepts messages of `HttpRequestEvent` type. Fortunately, the design flaw is exposed when upgrading the SOCKO implementation using TAkka.

6.3 Efficiency, Throughput, and Scalability

The TAkka library is built on top of Akka so that code for shared features can be re-used. The three main source of overheads in the TAkka implementation are: (i) the cost of adding an additional operation layer on top of Akka code, (ii) the cost of constructing type de-

Source	Example	Description	number of actor classes
Quviq [Arts et al. 2006]	ATM simulator	A bank ATM simulator with backend database and frontend GUI.	5
	Elevator Controller	A system that monitors and schedules a number of elevators.	6
Akka Documentation [Typesafe Inc. (b) 2012]	Ping Pong	A simple message passing application.	2
	Dining Philosophers	A application that simulates the dining philosophers problem using Finite State Machine (FSM) model.	2
	Distributed Calculator	An application that examines distributed computation and hot code swap.	4
	Fault Tolerance	An application that demonstrates how system responses to component failures.	5
Other Open Source Akka Applications	Barber Shop [Zachrison 2012]	A application that simulates the Barber Shop problem.	6
	EnMAS [Doyle and Allen 2012]	An environment and simulation framework for multi-agent and team-based artificial intelligence research.	5
	Socko Web Server [Imtarnasan and Bolton 2012]	lightweight Scala web server that can serve static files and support RESTful APIs	4
	Gatling [Excilys Group 2012]	A stress testing tool.	4
	Play Core [Typesafe Inc. (c) 2013]	A Java and Scala web application framework for modern web application development.	1

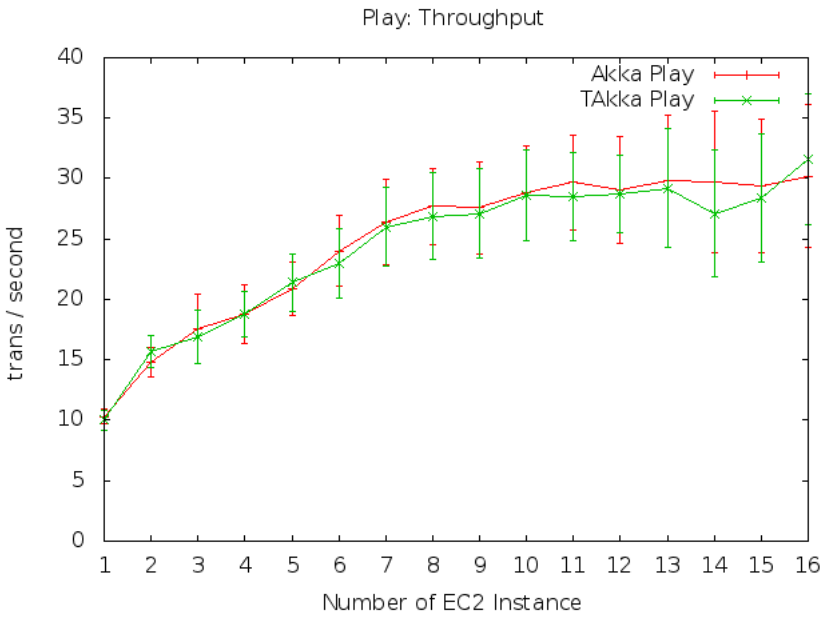
Table 2. Examples for Correctness and Expressiveness Evaluation

Source	Example	Akka Code Lines	Modified TAkka Lines	% of Modified Code	TAkka Code Lines	% of Code Size
Quviq	ATM simulator	1148	199	17.3	1160	101
	Elevator Controller	2850	172	9.3	2878	101
Akka Documentation	Ping Pong	67	13	19.4	67	100
	Dining Philosophers	189	23	12.1	189	100
	Distributed Calculator	250	43	17.2	250	100
	Fault Tolerance	274	69	25.2	274	100
Other Open Source Akka Applications	Barber Shop	754	104	13.7	751	99
	EnMAS	1916	213	11.1	1909	100
	Socko Web Server	5024	227	4.5	5017	100
	Gatling	1635	111	6.8	1623	99
	Play Core	27095	15	0.05	27095	100
geometric mean		991.7	71.6	7.4	992.1	100.0

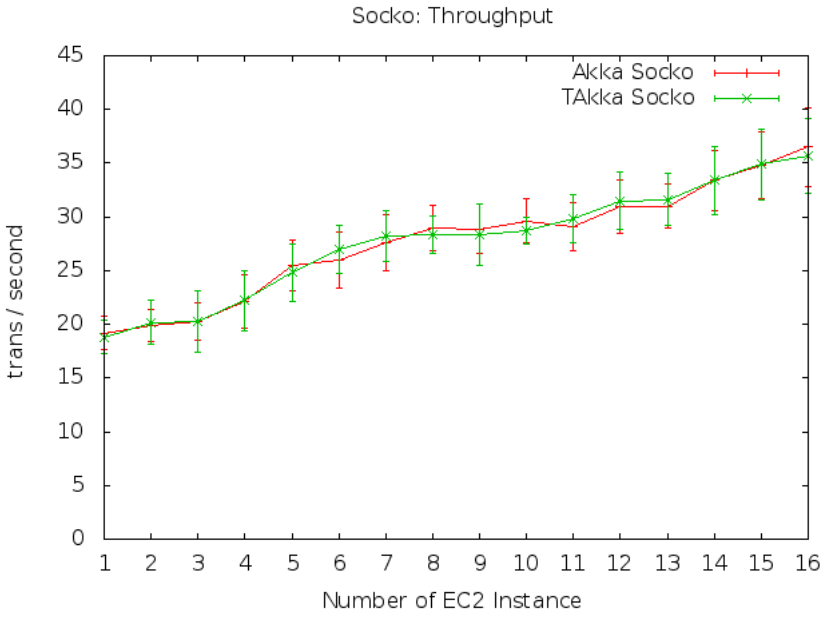
Table 3. Results of Correctness and Expressiveness Evaluation

scriptors, and (iii) the cost of transmitting type descriptor in distributed settings. We assess the upper bound of the cost of the first two factors by a micro benchmark which assesses the time of initializing n instances of `MyActor` defined in Figure 2 and Figure 7. When n ranges from 10^4 to 10^5, the TAkka implementation is about 2 times slower as the Akka implementation. The cost of the last factor is close to the cost of transmitting the string representation of fully qualified type names.

The JSON serialization example [TechEmpower, Inc. 2013] is used to compare the throughput of 4 web services built using Akka Play, TAkka Play, Akka Socko, and TAkka Scoko. For each HTTP request, the example gives an HTTP response with pre-defined content.



(a)



(b)

Figure 16. Throughput Benchmarks

Example	Description
bang	This benchmark tests many-to-one message passing. The benchmark spawns a specified number sender and one receiver. Each sender sends a specified number of messages to the receiver.
big	This benchmark tests many-to-many message passing. The benchmark creates a number of actors that exchange ping and pong messages.
ehb	This is a benchmark and stress test. The benchmark is parameterized by the number of groups and the number of messages sent from each sender to each receiver in the same group.
mbrot	This benchmark models pixels in a 2-D image. For each pixel, the benchmark calculates whether the point belongs to the Mandelbrot set.
ran	This benchmark spawns a number of processes. Each process generates a list of ten thousand random integers, sorts the list and sends the first half of the result list to the parent process.
serialmsg	This benchmark tests message forwarding through a dispatcher.

Table 4. Examples for Efficiency and Scalability Evaluation

All web services are deployed to Amazon EC2 Micro instances (t1.micro), which has 0.615GB Memory. The throughput is tested with up to 16 EC2 Micro instances. For each number of EC2 instances, 10 rounds of throughput measurement are executed to gather the average and standard derivation of the throughput. The results reported in Figure 16 shows that web servers built using Akka-based library and TAkka-based library have similar throughput.

We further investigated the speed-up of multi-node TAkka applications by porting 6 micro benchmark examples, listed in Table 4, from the BenchErl benchmarks in the RELEASE project [Boudeville et al. 2012]. Each BenchErl benchmark spawns one master process and many child processes for a tested task. Each child process is asked to perform a certain amount of computation and report the result to the master process. The benchmarks are run on a 32 node Beowulf cluster at the Heriot-Watt University. Each Beowulf node comprises eight Intel 5506 cores running at 2.13GHz. All machines run under Linux CentOS 5.5. The Beowulf nodes are connected with a Baystack 5510-48T switch with 48 10/100/1000 ports.

Figure 17 and 18 reports the results of the BenchErl benchmarks. We report the average and the standard deviation of the run-time of each example. Depending on the ratio of the computation time and the I/O time, benchmark examples scale at different levels. In all examples, TAkka and Akka implementations have almost identical run-time and scalability.

6.4 Assessing System Reliability

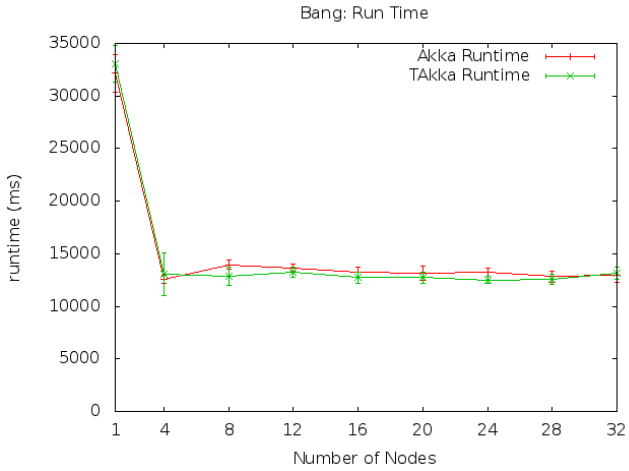
The supervision tree principle is adopted by Erlang and Akka users with the hope of improving the reliability of software applications. Apart from the reported nine ”9”s reliability of the Ericsson AXD 301 switch [Armstrong, Joe 2002] and the wide range of Akka use cases, how can software developers assure the reliability of their newly implemented applications?

TAkka is shipped with a Chaos Monkey library and a Supervision View library for assessing the reliability of TAkka applications. A Chaos Monkey test randomly kills actors in a supervision tree and a Supervision View test dynamically captures the structure of supervision trees. With the help of Chaos Monkey and Supervision View, users can visualize how their TAkka applications react to adverse conditions. Missing nodes in the supervision tree (Section 6.4.2) show that failures occur during the test. On the other hand, any failed actors are restored, and hence appropriately supervised applications (Section6.4.3) pass Chaos Monkey tests.

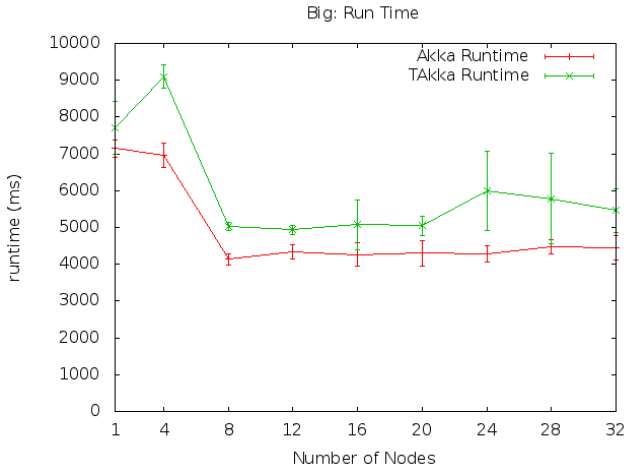
6.4.1 Chaos Monkey and Supervision View

A Chaos Monkey test [Netflix, Inc. 2013] randomly kills Amazon EC2 instances in an Auto Scaling Group. In a Chaos Monkey test, the reliability of an application is tested against intensive adverse conditions. Chaos Monkey is ported into Erlang to detect potential flaws of supervision trees [Luna 2013]. We port the Erlang version of Chaos Monkey into the TAkka library. In addition to randomly killing actors, users can simulate other common failures by using other modes in Table 5.

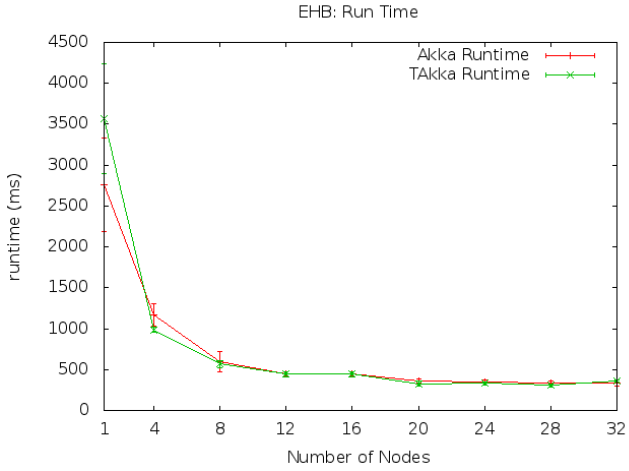
To dynamically monitor changes of supervision trees, we design and implement a Supervision View library. In a supervision view test, an instance of `ViewMaster` periodically sends request messages to interested actors. At the time when the request message is received, an active TAkka actor replies wih its status to the `ViewMaster` instance and passes the request message to its children. The status message includes its actor path, the paths of its children, and the time when the reply is sent. The `ViewMaster` instance records status messages and passes them to a visualizer, which will analyze and interpret changes of the tree structure



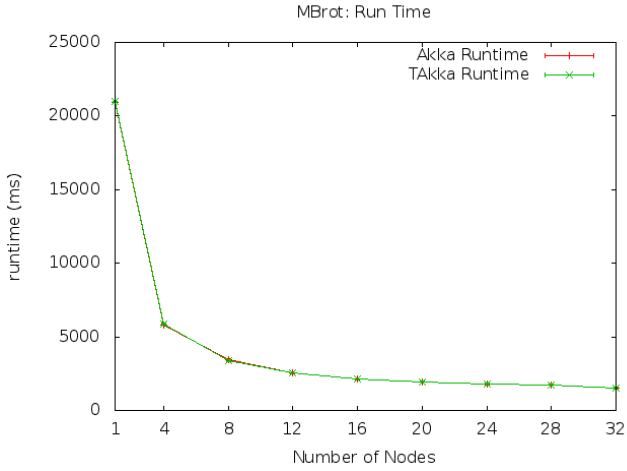
(a)



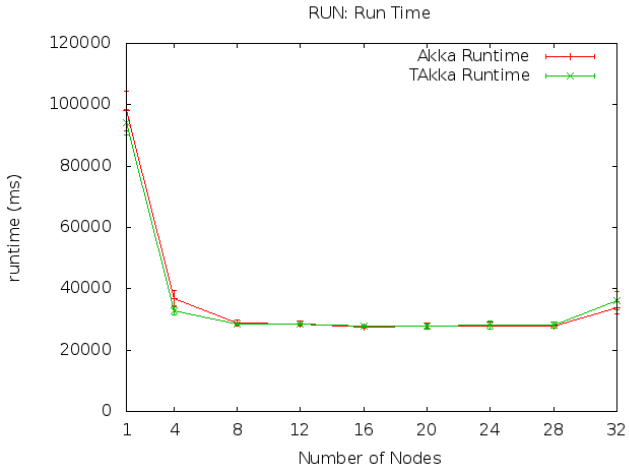
(b)



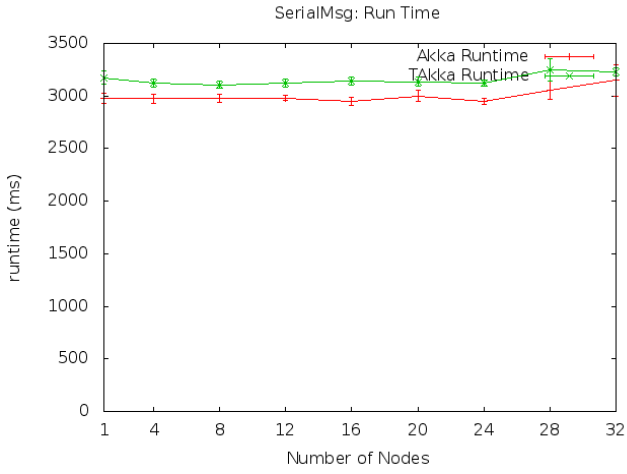
(c)



(d)

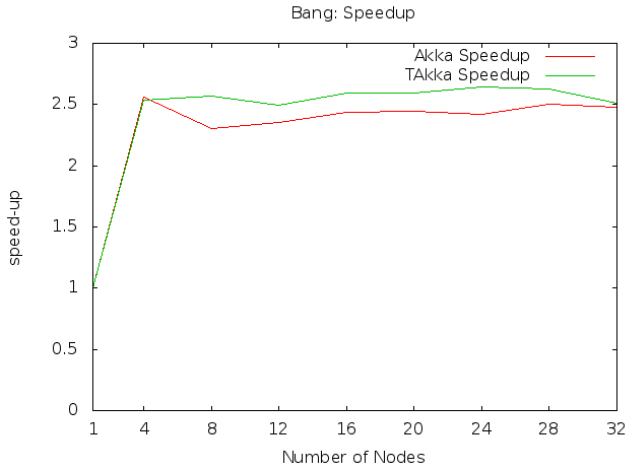


(e)

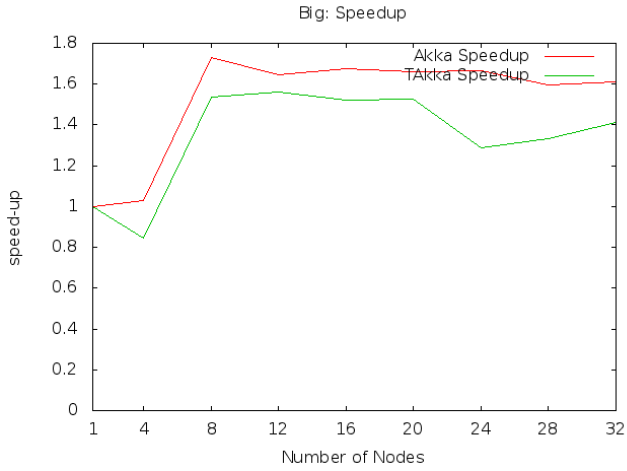


(f)

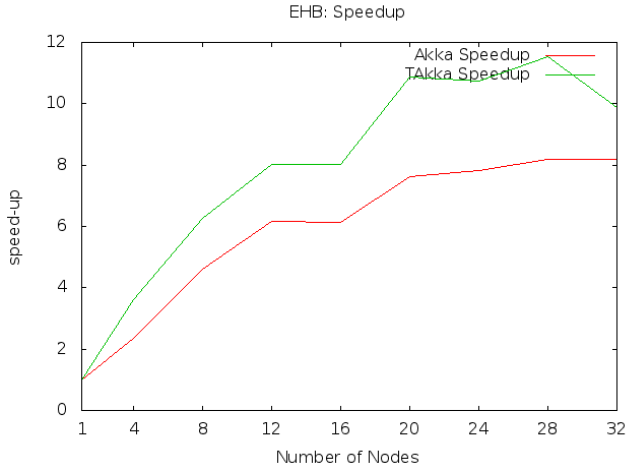
Figure 17. Runtime Benchmarks



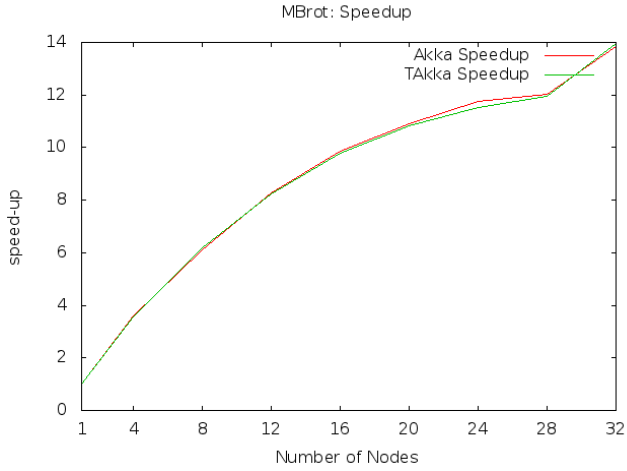
(a)



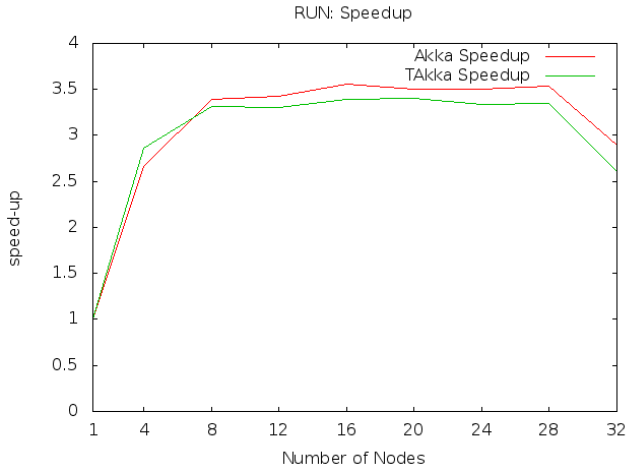
(b)



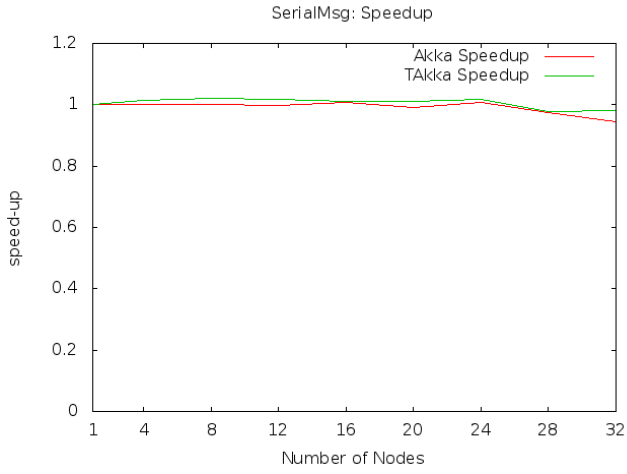
(c)



(d)



(e)



(f)

Figure 18. Scalability Benchmarks

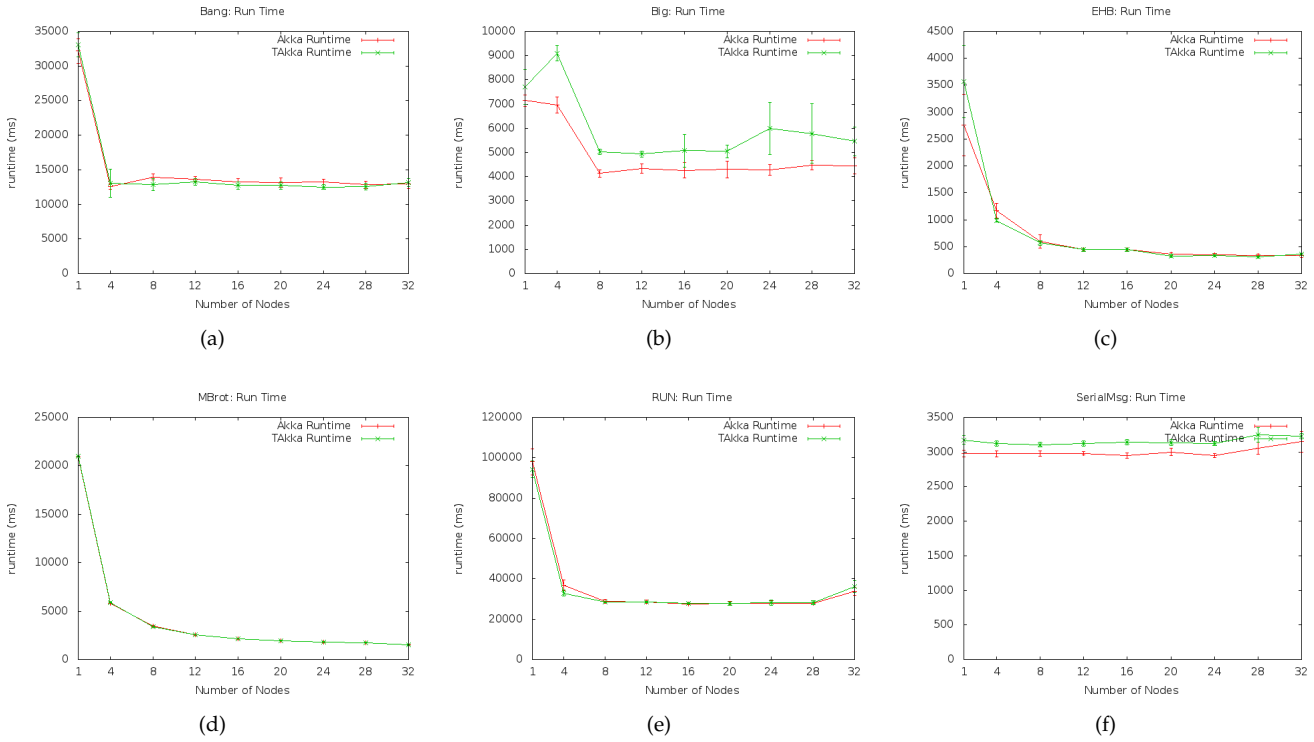


Figure 19. Runtime Benchmarks 2

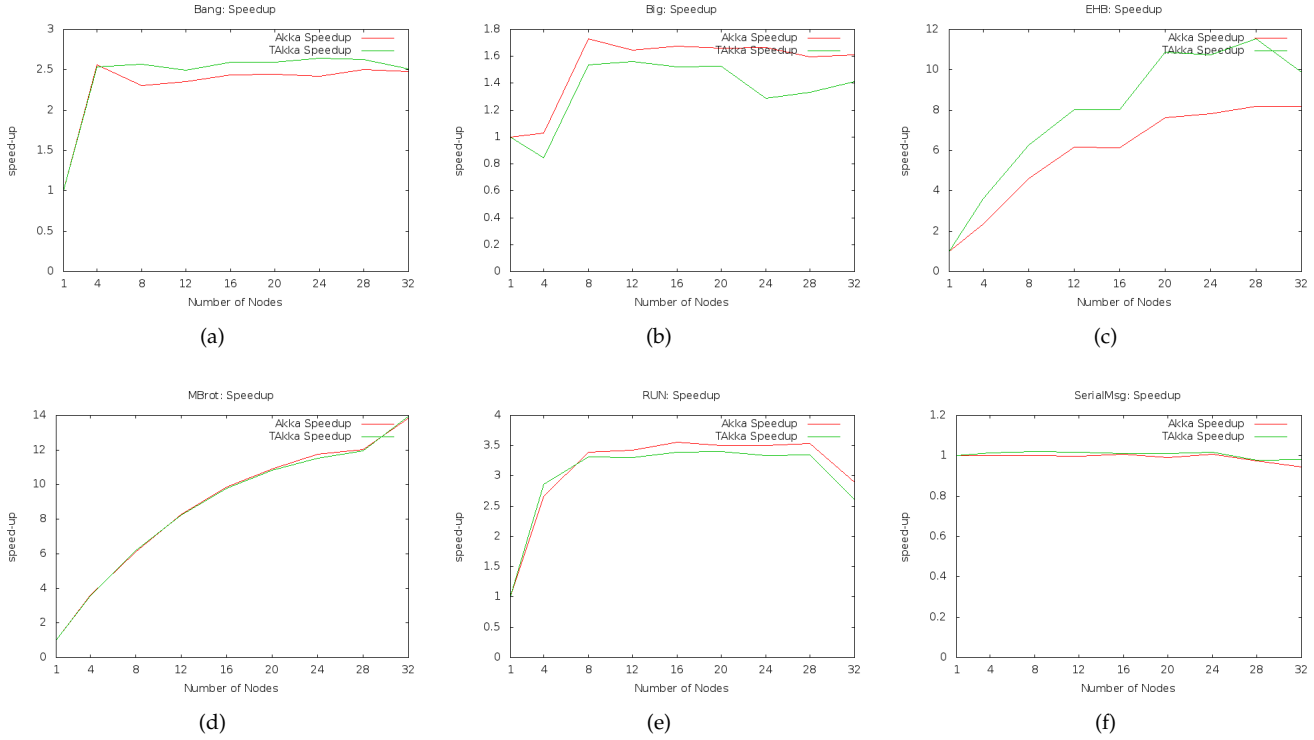


Figure 20. Scalability Benchmarks 2

Mode	Failure	Description
Random (Default)	Random Failures	Randomly choose one of the other modes in each run.
Exception	Raise an exception	A victim actor randomly raise an exception from a user-defined set of exceptions.
Kill	Failures that can be recovered by scheduling service restart	Terminate a victim actor. The victim actor can be restarted later.
PoisonKill	Unidentifiable failures	Permanently terminate a victim actor. The victim cannot be restarted.
NonTerminate	Design flaw or network congestion	Let a victim actor run into an infinite loop. The victim actor consumes system resources but cannot process any messages.

Table 5. TAkka Chaos Monkey Modes

during the testing period. We believe that similar library can be straightforwardly implemented in actor systems such as Akka and Erlang. From a practical point of view, the dynamically actor monitoring overcomes the two limitations of the static analyses tool developed in Nystrm's PhD thesis [Nyström 2009], which only applies to applications built using the Erlang/OTP library and cannot tell whether an actor will actually be started at runtime.

6.4.2 A Partly Failed Safe Calculator

In the hope that Chaos Monkey and Supervision View tests can reveal breaking points of a supervision tree, we modify the Safe Calculator example and run a test as follows. Firstly, we run three safe calculators on three Beowulf nodes, under the supervision of a root actor using the `OneForOne` strategy with `Restart` action. Secondly, we set different supervisor strategies for each safe calculator. The first safe calculator, S1, restarts any failed child immediately. This configuration simulates a quick restart process. The second safe calculator, S2, computes a Fibonacci number in a naive way for about 10 seconds before restarting any failed child. This configuration simulates a restart process which may take a noticeable time. The third safe calculator, S3, stops the child when it fails. Finally, we set-up the a Supervision View test which captures the supervision tree every 15 seconds, and a Chaos Monkey test which tries to kill a random child calculator every 3 seconds.

A test result, given in Figure 21, gives the expected tree structure at the beginning, 15 seconds and 30 seconds of the test. Figure 21(a) shows that the application initialized three safe calculators as described. In Figure 21(b), S2 and its child are marked as dashed circles because it takes the view master more than 5 seconds to receive their responses. From the test result itself, we cannot tell whether the delay is due to a blocked calculation or a network congestion. Comparing to Figure 21(a), the child of S3 is not shown in Figure 21(b) and Figure 21(c) because no response is received from it until the end of the test. When the test ends, no response to the last request is received from S2 and its child. Therefore, both S2 and its child are not shown in Figure 21(c). S1 and its child appear in all three Figures because either they never fail during the test or they are recovered from failures within a short time.

6.4.3 BenchErl Examples with Different Supervisor Strategies

To test the behaviour of applications with internal states under different supervisor strategies, we apply the `OneForOne` supervisor strategy with different failure actions to the 6 BenchErl examples and test those examples using Chaos Monkey and Supervision View. The master node of each BenchErl test is initialized with an internal counter. The internal counter decrease when the master node receives a finishing messages from its children. The test application stops when the internal counter of the master node reaches 0. We set the Chaos Monkey test with the `Kill` mode and randomly kill a victim actor every second. When the `Escalate` action is applied to the master node, the test stops as soon as the first `Kill` message sent from the Chaos Monkey test. When the `Stop` action is applied, the application does not stop and, eventually, the supervision view test only receives messages from the master node. When the `Restart` action is applied, the application does not stop but the Supervision View test receives messages from the master node and its children. When the `Resume` action is applied, all tests stops eventually with a longer run-time comparing to tests without Chaos Monkey and Supervision View tests.

7. Conclusion

Existing actor libraries accept dynamically typed messages. The TAkka library introduces a type-parameter

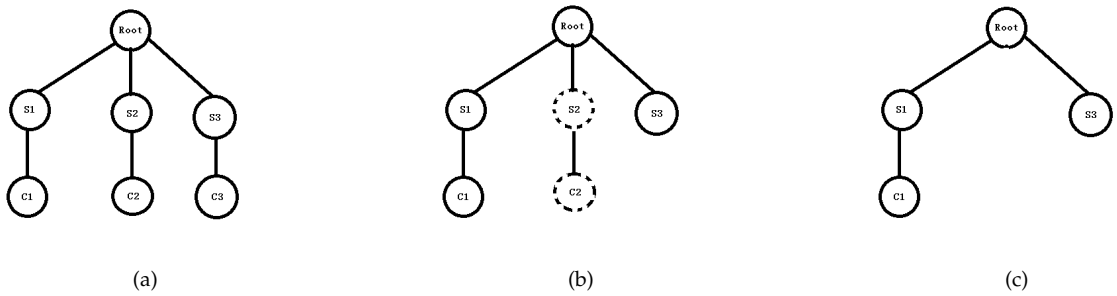


Figure 21. Supervision View Example

for actor-related classes. The additional type-parameter of a TAkka actor specifies the communication interface of that actor. With the help of type-parameterized actors, unexpected messages to actors are rejected at compile time.

In addition to eliminating programming bugs and type errors, programmers would like to have a failure recovery mechanism for unexpected run-time errors. We are glad to see that type-parameterized actors can form supervision trees in the same way as untyped actors.

Lastly, test results show that building type-parameterized actors on top of Akka does not introduce significant overheads, with respect to program size, efficiency, and scalability. In addition, debugging techniques such as Chaos Monkey and Supervision View can be applied to applications built using actors with supervision trees. The above results encourage the use of types and supervision trees to implement reliable applications and improve the reliability of legacy applications with little effort. We expect similar results can be obtained in other actor libraries such as future extensions of CloudHaskell [Watson et al. 2012].

Acknowledgments

Acknowledgments

References

J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Armstrong, Joe. Concurrency Oriented Programming in Erlang. `http://ll2.ai.mit.edu/talks/armstrong.pdf`, 2002.

T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. doi: 10.1145/1159789.1159792.

O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger. Release: a high-level paradigm for reliable large-scale server software. *Symposium on Trends in Functional Programming*, July 2012.

C. Doyle and M. Allen. EnMAS: A new tool for multi-agent systems research and education. *Midwest Instruction and Computing Symposium*, 2012.

J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034690.

Ericsson AB. Erlang Programming Language. `http://www.erlang.org`, 2012a. Accessed on Oct 2012.

Ericsson AB. OTP Design Principles User's Guide. `http://www.erlang.org/doc/pdf/otp-system-documentation.pdf`, 2012b.

Excilys Group. Gatling: stress tool. http://gatling-tool.org/, 2012. Accessed on Oct 2012.

C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/564585.564601.

P. Haller and M. Odersky. Event-Based Programming without Inversion of Control. In D. E. Lightfoot and C. A. Szyperski, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 4–22, 2006.

P. Haller and M. Odersky. Actors that Unify Threads and Events. In J. Vitek and A. L. Murphy, editors, *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, Lecture Notes in Computer Science (LNCS), pages 171–190. Springer, 2007.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, Sept. 2006. ISBN 0123704901.

C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*,

IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

V. Imtarnasan and D. Bolton. SOCKO Web Server. http://sockoweb.org/, 2012. Accessed on Oct 2012.

R. Kuhn, J. He, P. Wadler, J. Bonér, and P. Trinder. Typed akka actors. private communication, 2012.

D. Luna. Erlang Chaos Monkey. `https://github.com/dLuna/chaos_monkey`, 2013. Accessed on Mar 2013.

M. Naftalin and P. Wadler. *Java Generics and Collections*, chapter Chapter 5: Evolution, Not revolution. O'Reilly Media, Inc., 2006. ISBN 0596527756.

Netflix, Inc. Chaos Home. `https://github.com/Netflix/SimianArmy/wiki/Chaos-Home`, 2013. Accessed on Mar 2013.

J. H. Nyström. *Analysing Fault Tolerance for ERLANG Applications*. PhD thesis, ACTA UNIVERSITATIS UPSALIENSI, 2009.

D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521781779.

TechEmpower, Inc. Techempower web framework benchmarks. http://www.techempower.com/benchmarks/, 2013. Accessed on July 2013.

Typesafe Inc. (a). Akka API: Release 2.0.2. http://doc.akka.io/api/akka/2.0.2/, 2012. Accessed on Oct 2012.

Typesafe Inc. (b). Akka Documentation: Release 2.0.2. http://doc.akka.io/docs/akka/2.0.2/Akka.pdf, 2012. Accessed on Oct 2012.

Typesafe Inc. (c). Play 2.2 documentation. http://www.playframework.com/documentation/2.2-SNAPSHOT/Home, 2013. Accessed on July 2013.

T. Watson, J. Epstein, S. P. Jones, and J. He. Supporting libraries (a la otp) for cloud haskel. private communication, 2012.

M. Zachrison. Barbershop. https://github.com/cyberzac/BarberShop, 2012. Accessed on Oct 2012.