Type-parameterized Actors and Their Supervision

No Author Given

No Institute Given

Abstract. The robustness of actor-based concurrent 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. This paper introduces TAkka, a typed Akka library that supports both typed messages and the supervision principle. The TAkka library mixes statical and dynamical type checking to make sure that dynamically typed distributed resources and statically typed local resources have consistent types. Our notion of typed actor can publish itself as different types when used by different parties so that messages of unexpected types are prevented at the senders' side. In TAkka, messages for supervision purposes are treated specially so that a supervisor can interact with child actors of different types. Results show that Akka programs can be gradually upgraded to TAkka equivalents with minimal runtime and code size overheads. Finally, we implement two auxiliary libraries for reliability assessment.

1 Introduction

The Actor model defined by 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 [Armstrong, 2007] for building real time telephony applications. Erlang developers later propose the supervision principle [Ericsson AB., 2012], 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 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. Some recent actor libraries, including Cloud Haskell [Epstein et al., 2011], Lift [Typelevel ORG, 2013] , and scalaz [WorldWide Conferencing, LLC, 2013], support both dynamically and statically typed messages, but do 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.

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. [2013] and Luna [2013], this paper makes following contributions.

- It presents the design and implementation of a typed name server that maps typed names to values of the corresponding type, for example actor references. 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 language support for type reflection and 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 5 explains how to upgrading Akka programs to their TAkka equivalents gradually.
- It gives a comprehensive evaluation of the TAkka library. Section 6.1 shows that the TAkka library can avoid the type pollution problem straightforwardly. 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 explained in Section 6.4 are shipped with the TAkka library for assisting reliability evaluation. The first library, Chaos Monkey, is ported to test applications against intensive adverse conditions. The second library, Supervision View, is designed and implemented to dynamically monitor changes of supervision tree.

2 Actors and there Supervision in Akka

The Akka library [Typesafe Inc. (b), 2012] implements Actor programming in static typed language, Scala, and makes supervision obligatory. Figure 1 gives an example actor defined in Akka. The `receive` function of the Akka actor has type `Any⇒Unit` but the defined actor, `StringProcessor`, is only intended to process strings. At Line 14, a `Props`, an abstraction of actor creation, is initialized

and passed to an actor system, which creates an actor with name *processor* and returns a reference pointing to that actor. Another way to obtain an actor is using the `actorFor` method as shown in line 20. We then use actor references to send string messages and integer messages to the referenced actor. String messages are processed in the way defined by the receive function.

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

Fig. 1. Akka Example: A String Processor

Undefined messages, such as the two integer messages in the String Processor example, are treated differently in different actor libraries. In Erlang, an actor keeps undefined messages in its mailbox. It 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.

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.

Figure 2 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 that its child may not fail more than twice within a minute. If the child fails more frequently, the safe calculator itself will stop, and the 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 have been terminated since the simple calculator fails more frequently than allowed.

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 runtime 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

As in Java, generic types are erased by the Scala compiler. To record type information that is required at runtime but might be erased, Scala users can ask the compiler to keep the type information by using the `Manifest` class. The Scala standard library contains four manifest classes as shown in Figure

```
trait Operation
case class Multiplication(m:Int, n:Int) extends Operation
case class Division(m:Int, n:Int) extends Operation
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
  }
}
object SupervisedCalculator extends App {
  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
*/
```

Fig. 2. Akka Example: Supervised Calculator

3. A `Manifest[T]` encapsulates the runtime type representation of some type `T`. `Manifest[T]` a subtype of `ClassManifest[T]`, which declares methods for subtype (<:<) test and supertest (>:>). The object `NoManifest` represents type information that is required by a paraterized type but is not available in scope. `OptManifest[+T]` is the supertype of `ClassManifest[T]` and `OptManifest`.

```
package scala.reflect
trait OptManifest[+T] extends Serializable
object NoManifest extends OptManifest[Nothing] with Serializable
trait Manifest[T] extends ClassManifest[T] with Serializable
trait ClassManifest[T] extends OptManifest[T] with Serializable
  def <:<(that: ClassManifest[_]): Boolean
  def >:>(that: ClassManifest[_]): Boolean
  erasure: Class[_]
```

Fig. 3. Scala API: Manifest Type Hierarchy

```
case class TSymbol[T:Manifest](val s:Symbol) {
    private [takka] val t:Manifest[_] = manifest[T]
    override def hashCode():Int = s.hashCode()
}
case class TValue[T:Manifest](val value:T){
  private [takka] val t:Manifest[_] = manifest[T]
}
```

Fig. 4. TSymbol and TValue

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. `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 it can only be accessed by TAkka library developers. `TValue` is implemented similarly 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 symbolic 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 removes the entry *name* and returns true if (i) its symbolic 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 a registered name 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 `TValue` class does not appear in the APIs for library users, 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` may 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 the name server 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 support type reification, implementing typed name server can be difficult.

4 TAkka Library Design

4.1 Type-parameterized Actor

A TAkka actor has type `TypedActor[M]`. It inherits the Akka `Actor` trait to minimize the 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 interface 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
```

Fig. 5. TAkka API: TypedActor

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. 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]` in Scala. Consider rewriting the simple calculator defined in Figure 2 using TAkka, it is clear that `ActorRef` is contravariant 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]
```

Fig. 6. TAkka API: Actor Reference

For the ease of use, `ActorRef` provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method encapsulates the process of type cast on `ActorRef`, a contravariant type. We believe that using the notation of the `publishAs` method can be more intuitive than thinking about contravariance and subtyping relationship when publishing an actor reference as different types in a complex application. In addition, type conversion using `publishAs` is statically type checked. More importantly, 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. The last advantage is important in Scala because a library developer may not have access to code written by others.

Figure 7 defines the same string processing actor given in Figure 1. 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

```
class StringProcessor extends TypedActor[String] {
  def typedReceive = {
    case m:String => println("received message: "+m)
  }
}
object StringProcessorTest extends App {
  val system = ActorSystem("ServerTest")
  val processor = system.actorOf(Props[String, StringProcessor],
      "processor")

  processor ! "Hello World"
//  processor ! 1
// compile error: type mismatch; found : Int(3)
//   required: String

  val processorString = system.actorFor[String]
                  ("akka://ServerTest/user/processor")
  processorString ! "Hello World Again"
  val processorInt = system.actorFor[Int]
                ("akka://ServerTest/user/processor")
  processorInt ! 2
}
/*
Terminal output:
received message: Hello World
received message: Hello Worldv Again
Exception in thread "main" java.lang.Exception:
 ActorRef[akka://ServerTest/user/server] does not
 exist or does not have type ActorRef[Int] at
 takka.actor.ActorSystem.actorFor(ActorSystem.scala:223)
 ...
*/
```

Fig. 7. TAkka Example: A String Processor

`MyActor`, which is a subtype of `TypedActor[M]`, a Prop of type `Prop[M]` can be created by one of the code 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 using the `setReceiveTimeout` method, and (iv) update its behaviours using the `become` method. Compared with corresponding

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

Fig. 8. TAkka Example: Creating Actor Props

Akka APIs, our methods take an additional type parameter whose meaning will be explained below.

```
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]
```

Fig. 9. TAkka API: Actor Context

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, 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 describes the technique to replace system components without shutting down the system or causing significant interruption to the system. Hot swapping is a desired feature of distributed systems, whose components are typically developed and deployed 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 and the system message handler, those two 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, as the handler for system messages are separated from the handler for other messages, TAkka users may update the system message handler as well. Secondly, hot swapping in TAkka *must* be backward compatible and *cannot* be rolled back. 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.

```
abstract class ActorContext[M:Manifest]
 implicit private var mt:Manifest[M] = manifest[M]
 def become[SupM >: M](
    newTypedReceive: SupM => Unit,
    newSystemMessageHandler:
            SystemMessage => Unit
 )(implicit smtTag:Manifest[SupM]):ActorRef[SupM]

case class BehaviorUpdateException(smt:Manifest[_], mt:Manifest[_])
    extends Exception(smt + "must be a supertype of "+mt+".")
```

Fig. 10. TAkka API: Hot Swapping

The become method is declared 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. 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 Reusing Akka Supervisor Strategies in TAkka

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. The RestForOne strategy is not implemented in Akka. The Akka library does not implement it neither because it is not required for applications that we have considered.

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
```

Fig. 11. Supervisor Strategies

4.6 Handling System Messages

Actors communicate with each other by sending messages. To organize actors, a special category of messages should be handled by all actors. In Akka, those messages are subclasses of the `PossiblyHarmful` trait. The TAkka library retains some of them as subclasses of the `SystemMessage` trait.

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 type `SystemMessage` to be the supertype of all messages for system maintenance purposes. The three Akka system messages retained in TAkka are listed below. Other Akka system messages that does not related to supervision are not included in TAkka.

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

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.

The next question is whether a system message should be handled by the library or by application developers. In Erlang and early versions of Akka, all system messages can be explicitly handled by developers in the `receive`

block. In recent Akka versions, some system messages become private to library developers and some can be still handled by application developers.

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. Application developers 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.

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 1) 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 what we will do in Section 6.2, will the upgrade affect client code, especially legacy applications built using the Akka library? Fortunately, TAkka actors and actor references are implemented

```
class TAkkaStringActor 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[TAkkaStringActor], "aserver")

  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, TAkkaStringActor], "tserver")

  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
*/
```

Fig. 12. TAkka actor in Akka application

using inheritance and delegation respectively so that no changes are required for legacy applications.

TAkka actors inherits Akka actors. In Figure 12, the actor implementation is upgraded to the TAkka version as in Figure 7. The client code, line 135 through line 23, is the same as the old Akka version given in Figure 1. 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 29 in Figure 12, 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 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 13, we re-use the Akka actor, initialize the actor in an Akka actor system, and obtain an Akka actor reference as in Figure 1. Then, we initialize a TAkka actor reference, `takkaServer`, which only accepts `String` messages.

6 Library Evaluation

This section presents the 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 architecture 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].

```
1 class AkkaStringActor extends akka.actor.Actor {
2   def receive = { case m:String => println("received message: "+m) }
3 }
4 object AkkaInTAkka extends App {
5   val system = akka.actor.ActorSystem("AkkaSystem")
6   val akkaserver = system.actorOf(
7       akka.actor.Props[AkkaStringActor], "server")
8
9   val takkaServer = new takka.actor.ActorRef[String]{
10    val untypedRef = akkaserver
11  }
12  takkaServer ! "Hello World"
13 // takkaServer ! 3
14 // compile error: type mismatch; found : Int(3)
15 //  required: String
16 }
17 /*
18 Terminal output:
19 received message: Hello World
20 */
```

Fig. 13. Akka actor in TAkka application

TAkka solves the type pollution problem by using polymorphism. Take the code template in Figure 14 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 Tic-Tac-Toe example in the TAkka code repository.

6.2 Expressiveness and Correctness

Table 1 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

```
sealed trait ControllerMsg
class V2CMessage extends ControllerMsg
class M2CMessage extends ControllerMsg

trait C2VMessage
case class ViewSetController(controller:ActorRef[V2CMessage]) extends
C2VMessage
trait C2MMessage
case class ModelSetController(controller:ActorRef[M2CMessage]) extends
C2MMessage
class View extends TypedActor[C2VMessage] {
  private var controller:ActorRef[V2CMessage]
  // rest of implementation
}
Model extends TypedActor[C2MMessage] {
  private var controller:ActorRef[M2CMessage]
  // rest of implementation
}
class Controller(model:ActorRef[C2MMessage], view:ActorRef[C2VMessage])
     extends
TypedActor[ControllerMessage] {
  override def preStart() = {
    model ! ModelSetController( typedSelf.publishAs[M2CMessage])
    view ! ViewSetController( typedSelf.publishAs[V2CMessage])
  }
  // rest of implementation
}
```

Fig. 14. Template for Model-View-Controller

the overall code modification and code size by calculating the geometric mean of all examples. The evaluation results in Table 1 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 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 be-

Source	Example	Akka Code Lines	Modified TAkka Lines	% of Modified Code	TAkka Code Lines	% of Code Size
Quviq [Arts et al., 2006]	ATM simulator	1148	199	17.3	1160	101
	Elevator Controller	2850	172	9.3	2878	101
Akka Documentation [Typesafe Inc. (b), 2012]	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 [Zachrison, 2012]	754	104	13.7	751	99
	EnMAS [Doyle and Allen, 2012]	1916	213	11.1	1909	100
	Socko Web Server [Imtarnasan and Bolton, 2012]	5024	227	4.5	5017	100
	Gatling [Excilys Group, 2012]	1635	111	6.8	1623	99
	Play Core [Typesafe Inc. (c), 2013]	27095	15	0.05	27095	100
geometric mean		991.7	71.6	7.4	992.1	100.0

Table 1. Results of Correctness and Expressiveness Evaluation

cause 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.

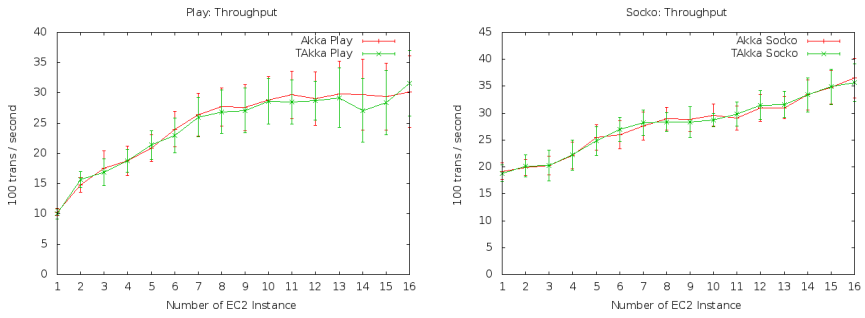
6.3 Efficiency, Throughput, and Scalability

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 descriptors, 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 1 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. 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 15 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, 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 16 and 17 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.



(a) (b)

Fig. 15. Throughput Benchmarks

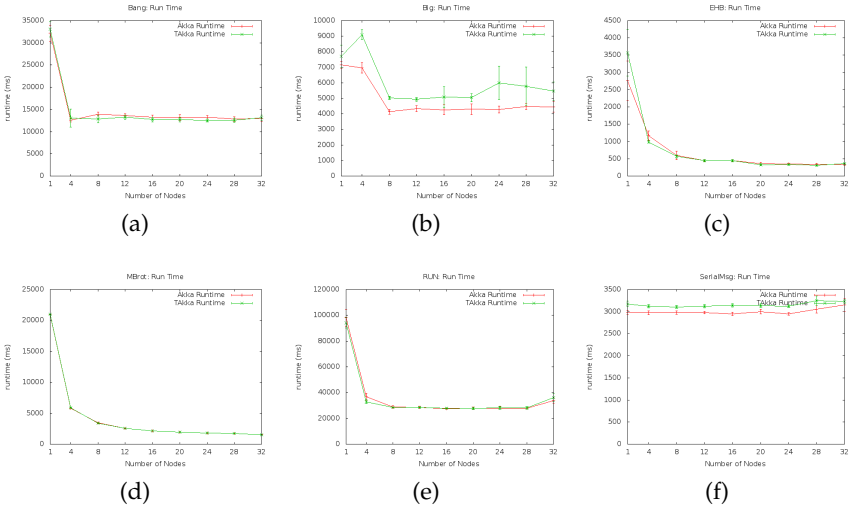


Fig. 16. Runtime Benchmarks 2

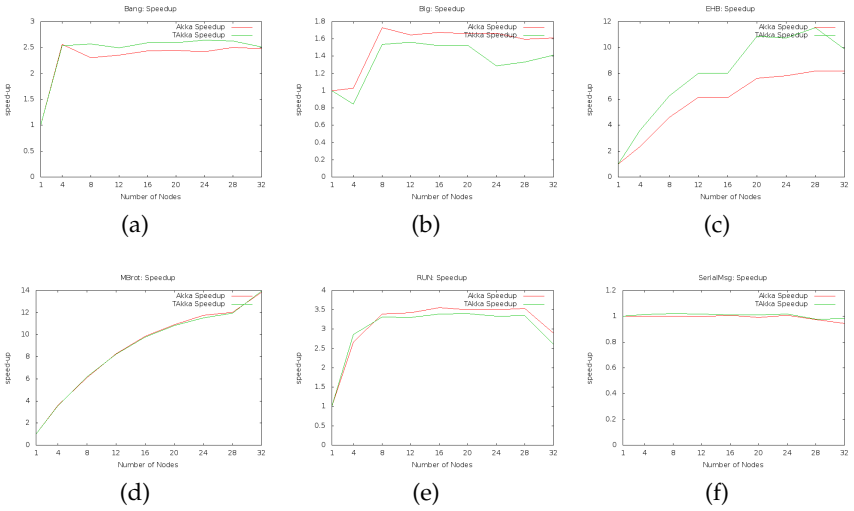


Fig. 17. Scalability Benchmarks 2

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. We expected that appropriately supervised actors will be restored when they fails, and hence pass Chaos Monkey tests.

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 2. TAkka Chaos Monkey Modes

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 2.

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 analyzes and interpret changes of the tree structure 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.

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 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].

Acknowledgements The authors gratefully acknowledge the substantial help they have received from many colleagues who have shared their related results and ideas with us over the long period during which this paper was in preparation. Benedict Kavanagh and Danel Ahman for continuous comments and discussions. The RELEASE team for giving us the access to the source code of the BenchErl benchmark examples. Thomas Arts from Quivq.com and Francesco Cesarini from Erlang Solutions for providing the Erlang source code for the ATM simulator example and the Elevator Controller example, two examples used in their commercial training courses.

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

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.

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.

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.

Typelevel ORG. scalaz: Functional programming for Scala. `http://typelevel.org/projects/scalaz/`, 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. Peyton Jones, and J. He. Supporting libraries (a la otp) for cloud haskell. private communication, 2012.

WorldWide Conferencing, LLC. Lift. `http://liftweb.net/`, 2013.

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