Distributed Programming with Types and Supervision Tree: Third Year PhD Progress Report

Jiansen HE

August 17, 2013

Abstract

This report reviews objectives and progress of my PhD project. Main research problems have been solved during the process of designing and implementing the TAkka library. The TAkka library has been evaluated in terms of correctness, expressiveness, and scalability. To help the design of applications with a high reliabilities, the TAkka library is also shipped with tools for debugging and monitoring purposes.

Statistic analysis shows that it is impractical to measure the reliabilities of a system with low failure rate. I propose to extend the scope of this research to include a formal specification of supervision tree. The specification gives a methodology to derive the reliability of a system built using supervision tree from the dependencies of nodes and the reliabilities of individual nodes.

1 Project Motivation and Objectives

The aim of this PhD project is to *study how types and the supervision principle can be merged to help the construction of reliable distributed systems* . This project intends to explore factors that encourage programmers to employ types and good programming principles. To this end, the following iterative and incremental tasks are set.

1. To identify some applications implemented in Erlang or Akka using the supervision principles. Those examples will be served as references to evaluate frameworks that support distributed programming. Candidate examples shall cover a wide range of aspects in distributed programming, including but not limited to (a) transmitting messages and potentially computation closures, (b) modular composition of distributed components, (c) default and extensible failure recovery mechanism, and (d) dynamic topology and hot code swapping.

2. To implement a library, TAkka, that supports actor programming and the supervision principles in a strongly typed setting. Basic components of the library may be built on top of Akka[**?**] in the case that repeated implementation could be avoided. Comparing to the Akka library, the new library should be able to prevent more forms of type errors at earlier development stages.
3. To re-implement identified applications using the library developed in task 2. The primary purpose of this task is to be aware of the strengths and limitations of using strongly and weakly typed supervision libraries. As a research which contains certain levels of overlaps with other existing and evolving frameworks, this research should distinguish itself from others by devised solutions to some problems which are difficult to be solved by alternatives.
4. To evaluate how different supervision libraries meet the demands of distributed programming. This project suggests evaluating libraries with respects to efficiency, scalability, and reliability.

2 Progress Review

The main topic of this research was set as the result of the background study in the first year. During the second year, the research objectives and the design of the TAkka library were refined in quarterly reviews. By the end of the second year, I had *a*) implemented key components of the TAkka library; *b*) proposed and implemented a novel typed nameserver that maps each typed symbol to typed a value of corresponding type; *c*) demonstrated how to solve the type pollution problem using type-parameterized actor and subtyping; and *d*) proposed a methodology for evaluating libraries supporting the supervision principle.

The third year is focused on the methodology for evaluating TAkka and other libraries that support the supervision principle. In short,

- The correctness and the expressiveness of the TAkka library is evaluated by porting a selection of small and medium sized applications written in Erlang and Akka.
- The scalability of sample applications written in TAkka and Akka is compared regarding to speed-up and throughput increases. Benchmark examples for speed-up measurement are selected from the Bencherl project and tested on the Beowulf cluster at the Heriot-Watt University. The example for throughput measurement is selected from the Techempower web framework benchmarks [**?**] and is tested using Amazon EC2 instances with a Load Balancer.
- The reliability of a TAkka application can be tested by using the ChaosMonkey library and the SupervisionView library. A chaos monkey test assesses the robustness of a TAkka application by randomly kills actors. The SupervisionView library can be used to monitor dynamical changes of a supervision tree.

The design, implementation, and evaluation of TAkka libraries are elaborated in the attached paper.

3 Measuring Reliability

Due to the nature of library development, we cannot assure the reliability of applications built using the supervision principle; nor the achieved high reliability of large Erlang applications can indicate that a newly implemented application using the supervision principle will have a desired reliability. To help software developers identifying bugs of their applications, the ChaosMonkey library and the SupervisionView library are shipped with TAkka. However, a quantitative measurement of software reliability under operational environment is still desired in practice. To solve this problem, two approaches are discussed in the attached research proposal.

The first approach is measuring the target system as black-box. Unfortunately, [**?**] shows that even a long time failure-free observation itself does not suggest the software system will achieve a high reliability in the future. I attempted to replace [**?**]'s single-run experiment by iterative experiment so that the required experiment time for some applications can be reduced. Unfortunately, in most cases, predicting the reliability of a system with low failure rate is still impractical. Therefore, I proposed the second approach which focuses on applications built using the supervision principle.

The second approach is giving a specification of actor-based supervision tree and measuring the reliability of a supervision tree as the accumulated result of reliabilities of its sub-trees. By eliminating language features not related to supervision, both the worker node and the supervisor node in a supervision tree can be modelled as Deterministic Finite Automata. Analysis shows that various of supervision trees can be modelled by a supervision tree that only contains simple worker node and simple supervisor node. More importantly, the reliability of a node in the proposed model is simply defined as the possibility that the node is in its **Free** state, in which it can react to a request. To accomplish this study, following problems need to be solved:

- What are possible dependencies between nodes? For each dependency, what is the algebraic relationship between the reliability of a sub-tree and reliabilities of individual nodes?
- Based on the above result, how to calculate the overall reliabilities of a supervision tree? When is the reliability is improved by using supervising tree, and when not?
- Given the reliabilities of individual workers and constraints between them, is there an algorithm to give a supervision tree which gives a desired reliability? If not, can we determine if the desired reliability is not achievable?

4 Timeline for Completing Researches

September 2013

- Complete the TAkka paper. The target venue will be ECOOP. The deadline for ECOOP14 Research Papers has not been announced. The deadline for ECOOP13 Research Papers is 22 Dec 2012.
- Write a technical memo about the proposed specification of supervision tree. The specification will focus on actor-based systems. Candidate Erlang and Akka examples will be identified to check the coverage of the specification.

December 2013

- Document the design, implementation, and evaluation of the TAkka library in thesis format.
- Complete the specification for supervision tree. The specification will includes a comprehensive study on the relationship between system reliability and common dependencies between actors.
- Write a brief proposal on extending the scope of the specification for supervision tree so that it can model a wider range of systems. The proposal will include the target scope of the generalisation, any modifications to the specification for actor-based system, and examples for the evaluation process.
- Write up a quarterly review report about the above tasks. The purpose of the quarterly review is tracking the direction and progress of my PhD research project. The review report will discuss whether extending the scope of the specification for supervision tree should be part of the PhD research.

March 2014

- Document the specification for supervision tree in thesis format.

June 2014

- Complete the thesis writing.

Type-parameterized Actors and Their Supervision (v6.5)

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 has been implemented in the OTP (Open Telecom Platform) library for Erlang and the Akka library for Scala. The later approach has been well explored in systems including the join-calculus and the typed π-calculus. An important open question is how easily the two approaches can be combined.

Combining the supervision principle with typed messages raises three challenges. Firstly, a novel name server is required to retrieve typed actor references for typed actor paths. Secondly, supervisor actors must interact with a wide variety of child actors and it is not obvious whether or not it would be easy to type supervisors. Thirdly, an actor which receives messages from distinct parties may suffer from the type pollution problem, in which case a party imports too much type information and can send the actor a message not expected from it.

This paper introduces the typed Akka library, TAkka, which resolves above problems. Although TAkka actors inherit from Akka actors and Scala TypeTag is used as serializable type information, we believe that similar improvements can be made to actor libraries in other languages.

We evaluate the TAkka library by re-implementing 19 examples built from Erlang or Akka libraries. Results show that TAkka adds little runtime and code size overheads to Akka. TAkka programs have similar scalability to their Akka equivalents. Finally, we port the Chaos Monkey library for testing the 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. The OTP Design principles, especially the supervision tree principle, made it easier to build reliable distributed applications [Armstrong 2007].

The notions of actor and supervision tree 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.

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 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 novel name server is required to retrieve actor references of a given 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 always to implement as a map from names to processes. Can this be adapted to the cases where processes are statically typed?

2. Supervisors must interact with a wide variety of processes. Actors are structured in supervision trees to improve system reliability. Each actor in a supervision tree needs to handle messages of its interest and a special category of messages for supervision purposes. Is it practical to define an actor that supervises children parameterized by a variety of types?
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-based component receives messages from distinct parties using its sole channel, when the actor-based component is naively parameterized with the union type of all expected message types, or receives dynamically typed messages like in Erlang and Akka designs, every party can send the actor messages not expected from it. Can a type-parameterised actor have different types when used by distinct 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 four 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 meets the requirement.
- It describes the design of the TAkka library. Section 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, section 4.5 and 4.6 shows 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 be 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 contravarient types.
- It gives a critical 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 initialisation stage but have almost identical run-time performance and scalability comparing to their Akka equivalents. The TAkka library also ships with a Chaos Monkey library and a Supervision View library for revealing breakpoint of supervision trees.

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 good programming practices in Object-Oriented Programming (OOP). The Release Handling principle requires runtime support on hot swapping. In a platform, for example the Java Virtual Machine (JVM), where hot swapping is not supported in general, hot swapping a particular component can be simulated by updating the reference to that component. For example, Section 4.4 explains how hot swapping the behaviour of an actor is supported in TAkka, which runs on the JVM. Table 1 lists the observed correspondence between OTP design principles and programming practices in JAVA and Scala. The Supervision Tree principle, which is the central topic of this paper, has no direct correspondence in native JVM based systems.

OTP Design Principle	JAVA/Scala Programming
Supervision Tree	no direct correspondence
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

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

2.2 Akka Actor

An Akka Actor has four important fields given 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. Line 16 creates and passes a `Props`, an abstraction of actor creation, to an actor system, 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 integer messages. String 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

```
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

integer message in the above short example, an event handler are defined and created with 8 lines of code.

2.3 Supervision

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

The Akka library makes supervision obligatory by restricting the way of creating actors. Actors can only

be initialised 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 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 restart the simple calculator when an `ArithmeticException` is raised. The supervisor strategy of the safe calculator also specifies the maximum failures its child may have within a 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 and fifth message are delivered. The last message is not processed because the both calculators a terminated because 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. However, 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 summarises the type reflection mechanism in Scala and explains how it benefits the implementation of our typed name server. Our typed name

```
1 case class Multiplication(m:Int, n:Int)
2 case class Division(m:Int, n:Int)
3
4 class Calculator extends Actor {
5   def receive = {
6     case Multiplication(m:Int, n:Int) =>
7       println(m +" * "+ n +" = "+ (m*n))
8     case Division(m:Int, n:Int) =>
9       println(m +" / "+ n +" = "+ (m/n))
10  }
11 }
12
13 class SafeCalculator extends Actor {
14   override val supervisorStrategy =
15     OneForOneStrategy(maxNrOfRetries = 2,
          withinTimeRange = 1 minute) {
16       case _: ArithmeticException =>
17         println("ArithmeticException Raised to:
              "+self)
18         Restart
19     }
20   val child:ActorRef =
         context.actorOf(Props[Calculator], "child")
21   def receive = {
22     case m => child ! m
23   }
24 }
25
26   val system = ActorSystem("MySystem")
27   val actorRef:ActorRef =
         system.actorOf(Props[SafeCalculator],
28 "safecalculator")
29
30   calculator ! Multiplication(3, 1)
31   calculator ! Division(10, 0)
32   calculator ! Division(10, 5)
33   calculator ! Division(10, 0)
34   calculator ! Multiplication(3, 2)
35   calculator ! Division(10, 0)
36   calculator ! Multiplication(3, 3)
37
38 /*
39 Terminal Output:
40 3 * 1 = 3
41 java.lang.ArithmeticException: / by zero
42 ArithmeticException Raised to:
       Actor[akka://MySystem/user/safecalculator]
43
44 10 / 5 = 2
45 java.lang.ArithmeticException: / by zero
46 ArithmeticException Raised to:
       Actor[akka://MySystem/user/safecalculator]
47 java.lang.ArithmeticException: / by zero
48
49 3 * 2 = 6
50 ArithmeticException Raised to:
       Actor[akka://MySystem/user/safecalculator]
51 java.lang.ArithmeticException: / by zero
52 */
```

Figure 3. Supervised Calculator

server can be straightforwardly ported to other platforms that support type reflection.

3.1 Scala Type Descriptors

Scala 2.8 introduces 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]]
```

3.2 Typed Name Server

In distributed systems, a name server maps each registered name, usually a unique string, to a 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 values retrieved from the name server is dynamically typed, values needs to be checked against and be cast to expected type by the client before use.

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.

[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.

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 the name `TSymbol` 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, we can implement a typed name server that provides following three operations:

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

  The 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 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 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 conforms the associated of the registered name. To permit polymorphism, the `hashcode` method of `TSymbol` defined in Figure 4 does not take type value 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 efficient library implementation because the type

information in `TSymbol` is neglected 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 rasing an exception so that it always availiable.

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 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 error 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 feature, implementing typed name server 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 from the Akka `Actor` trait to minimise implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor APIs. Instead, we encourage programmers to use Akka equivalent 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 reference pointing to itself.

```
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 Actor 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 counterpart or rewritten in TAkka. We believe that re-implementing the TAkka Actor library requires similar amount of work of 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 could 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 contravariant because `ActorRef[Operation]` is a subtype of `ActorRef[Division]` though `Division` is a subtype of `Operation`. Contraveriance 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 the supertype of the original actor reference. Secondly, the semantic 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 the same as our intention. In this example, types of `typedReceive` and `m` may be omitted because they could be inferred by the compiler. Unlike the Akka example, sending an integer to `server` is rejected by the compiler. Although the type error introduced at 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]` could be created by one of the APIs in figure 8:

Contrary to actor reference, 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 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. 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 created actor is assigned with a typed actor path, an Akka actor path together with a `Manifest` of the message type. We set up a typed name server in each actor system. When an actor is created, we register the mapping from the assigned typed actor path to typed actor reference in the typed name server inside the actor

```
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

system where the created actor resides. By doing this, the `actorFor` method in the `ActorContext` classes can return an actor reference of the desired type, if the actor located at the specified actor path has a compatible type.

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.

```
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

```
trait ActorContext[M] {
  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

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 as well. In the Akka 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 same amount of or more 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. 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 oder 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 require 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 throws the exception to the supervisor of the supervisor. The `Restart` action replaces the failed child with a new one. The `Resume` action asks the child to process the message again. The `Stop` action terminates the failed actor permanently.

None of the supervisor strategy in Figure 11 requires type-parameterized class to construct. Therefore, from the perspective of API design, both supervisor strategies are constructed in TAkka in the same way as in Akka.

4.6 Handling System Messages

Actors communicate with each other by sending messages. To maintain a supervision tree, a special category

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

Figure 11. Supervisor Strategies

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 a 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 system or by users. In Erlang and early Akka versions, all system messages could 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 to be handled by application developers. In TAkka, `ReceiveTimeout` is the only system message that can be explicitly handled by users.

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

Nevertheless, we keep the `SystemMessage` trait in the library so that new system messages can be included in the future if needed.

A key design decision in TAkka is to separate the system message and user-defined message handlers. The above decision has two benefits. Firstly, the type parameter of actor-related classes only denotes the type of user defined messages rather than the untagged union of user defined and system messages. Therefore, the TAkka API applies to systems that do not support an 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 could be supervised. Users of Akka typed actors invoke a service by calling a method instead of sending messages. Code in Figure 12 demonstrates how to define a simple string processor using Akka typed actor. Akka typed actors prevent some type errors but have two limitations. Firstly, Akka typed actors do not permit hot swapping on its behaviours. Secondly, avoiding type pollution by using Akka typed actors is as awkward as by using a plain object-oriented model, where supertypes need to be introduced. In Scala and Java, introduce a super type in a type hierarchy requires modification to all affected subtypes.

```
trait MyTypedActor{
  def processString(m:String)
}
class MyTypedActorImpl(val name:String) extends
     MyTypedActor{
  def this() = this("default")

  def processString(m:String) {
    println("received message: "+m)
  }
}
object FirstTypedActorTest extends App {
  val system = ActorSystem("MySystem")
  val myTypedActor:MyTypedActor =
    TypedActor(system).typedActorOf(
    TypedProps[MyTypedActorImpl]())
  myTypedActor.processString("Hello World")
}

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

Figure 12. Akka TypedActor Example

Actors with or without Mutable States The actor model formalised 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 ♠**Erlang actors may evolve to new actors with a different state value** ♠. 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 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 efficient fault-tolerant services. The CAP theorem [Gilbert and Lynch 2002] states it is impossible to achieve consistency, availability, and partition tolerance in a distributed system simultaneously. For actors that contain mutable states, system providers have to either sacrifice availability or partition tolerance, or modify the consistency model. For example, Akka actors have mutable state and Akka cluster developers spend a great effort to implement an eventual consistency model [Kuhn et al. 2012]. If an actor can define its behaviour without using mutable states, as evidenced by the success of RESTful web services, the actor is more likely to have a good scalability and availability.

Bi-linked Actors In addition to one-way linking in the supervision tree, Erlang and Akka provide mechanism to define two-way linkage between actors. Bi-linked actors are aware of the liveness of each other. We consider bi-linked actors is a redundant design 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 organised 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 equivalent generic version (evolution), rather than requiring use 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 use an Akka actor in a TAkka system.

5.1 TAkka actor in Akka system

It is often the the case that an actor-based library is implemented by one organisation but used in a client application implemented by another organisation. If a developer decided to upgrade the library implementation using TAkka actors, for example, 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 would the upgrade affects client code, especially legacy applications built using the Akka library? TAkka actor and actor reference 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 as 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 below, we get an untyped actor reference from `typedserver` and use the untyped actor reference in code where an Akka actor reference is expected. Because untyped actor reference accepts messages of any type, messages of unexpected may be sent to TAkka actors if Akka actor reference is used. As a result, users who are interested in the `UnhandledMessage` event need to subscribe the event stream as in line 33.

5.2 Akka Actor in TAkka system

Sometimes, developers want to update client code before actor is upgraded. For example, a developer may not have the access to the actor code; or the library may be large, so the developer may want to upgrade the library gradually.

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

6. Library Evaluation

This section presents preliminary evaluation results of the TAkka library. We show that the Wadler's type pollution problem can be avoided in a straightforward way

```
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

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 bringing 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

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].

Another solution to the type pollution problem is to use 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 model actor.

Therefore, both the view and the model only know the communication interface between the 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 21 and line 22 could be omitted, we explicitly use the publishAs to express our intentions and let the compiler check the type safety. The Tik-Tak-Tok example in the TAkka code repository is developed from the given code template.

```
1  sealed trait ControllerMsg
2  class V2CMessage extends ControllerMsg
3  class M2CMessage extends ControllerMsg
4
5  trait C2VMessage
6  fcase 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

6.2 Expressiveness and Correctness

Table 2 lists 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 in TAkka. We assess the overhead of code modification and code size by calculating the geometric mean of all examples, according to the methodology suggested by Hennessy and Patterson [Hennessy and Patterson 2006]. We give the evaluation results in Table 3. Results 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 because the designer always passes instances of `HttpRequestEvent` to the `unapply` method and send the the returned values to an actor that accepts messages of `HttpRequestEvent` type. Fortunately, the design flaw is exposed by the compiler when upgrading the SOCKO implementation using TAkka.

6.3 Performance, 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 descriptor, 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.

We use the JSON serialization example [TechEmpower, Inc. 2013] to compared 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. Web services are deployed to Amazon EC2 instances

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

with a Load Balancer. All EC2 instances are Micro instances (t1.micro) with 0.615GB Memory. We use shell script to spawn 1200 siege tests [Jeff Fulmer 2013] in parallel. Each siege test sends 500 requests in its benchmark mode. The result summarised in Figure 20 shows the throughput is approximately linear to the number of EC2 instances, before we reaching the limit of the bandwidth.

We further investigated the efficiency and scalability of TAkka 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

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

many child processes for a tested task. Each child process is asked to perform a certain amount of calculation and report the result to the master process. The benchmarks are run on a Beowulf cluster at the Heriot-Watt University. The 32 Beowulf cluster nodes each comprise 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 the BenchErl benchmarks. We report the average and the standard deviation of the run-time of each example. Depending on the ratio of the calculation time and the I/O time, benchmark examples scales 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 Ericsson AXD 301 switch [Armstrong, Joe 2002] and the wide range of Akka use cases, how could 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

Chaos Monkey [Netflix, Inc. 2013] randomly kills Amazon Elastic Compute Cloud (Amazon EC2) instances in an Auto Scaling Group. In a Chaos Monkey test, the reliability of an application is tested against an intensive adverse conditions. The same idea 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 of a target TAkka application. On the time when the request message is received, an active TAkka actor replies its status to the view master and pass the request message to its children. The status message includes its actor path, paths of its children, and the time when the reply is sent. The view master records received status messages and passes them to a visualiser, which analyzes and interprets changes of the tree structure during the testing period, at the end of the test.

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, restart its failed child immediately. The second safe calculator, S2, computes a Fibonacci number in a naive way for about 10 seconds before restarting failed child.

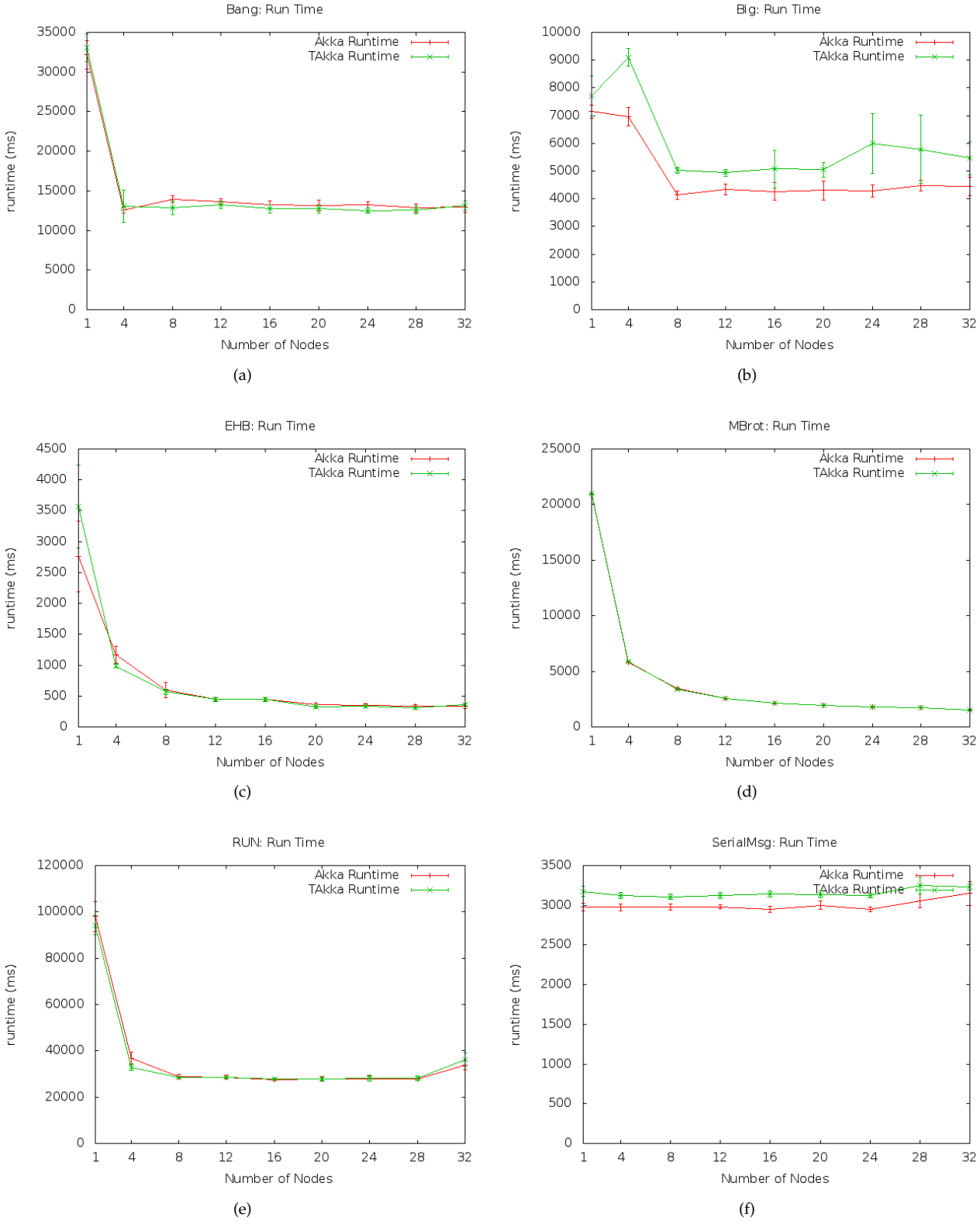
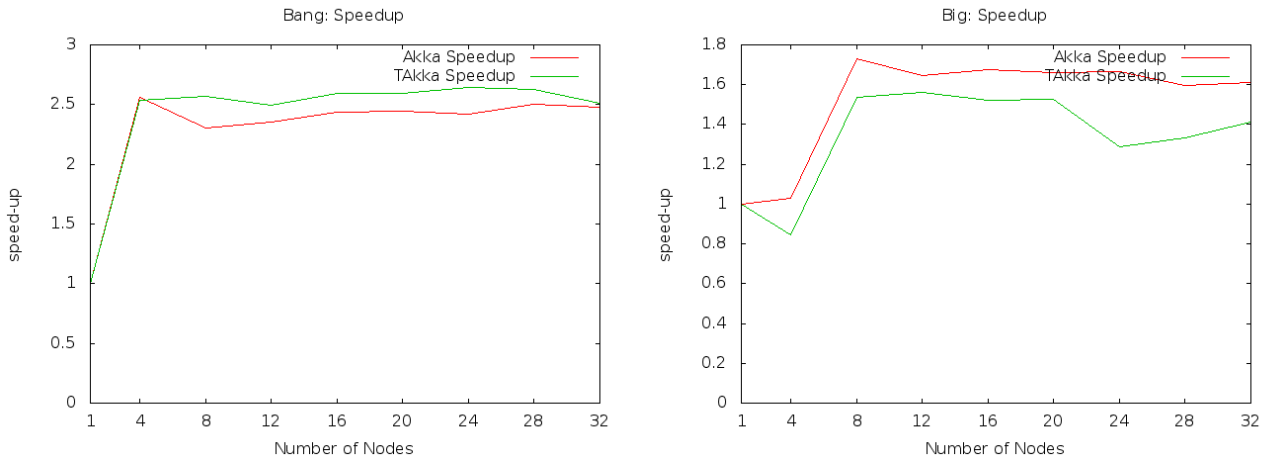
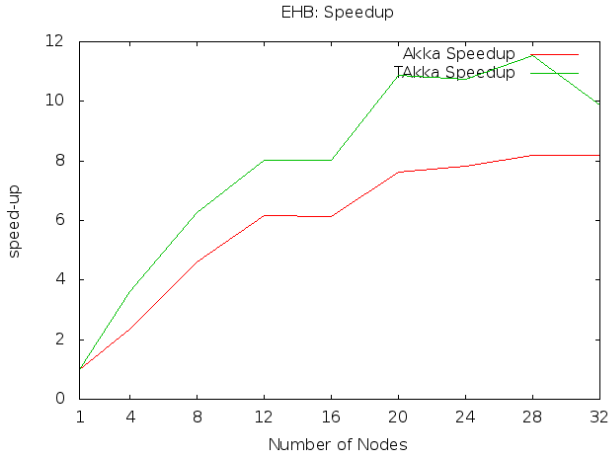


Figure 16. Runtime Benchmarks

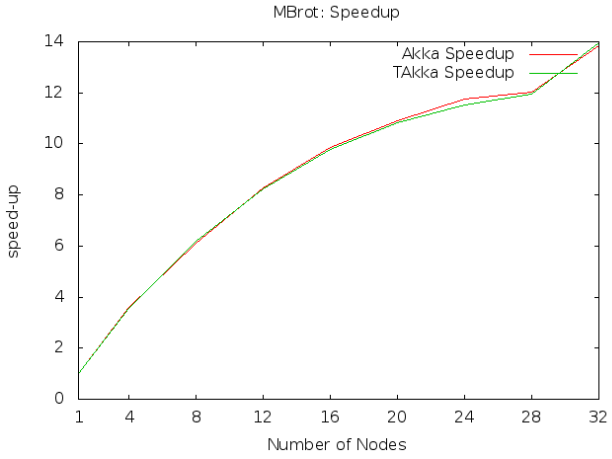


(a)

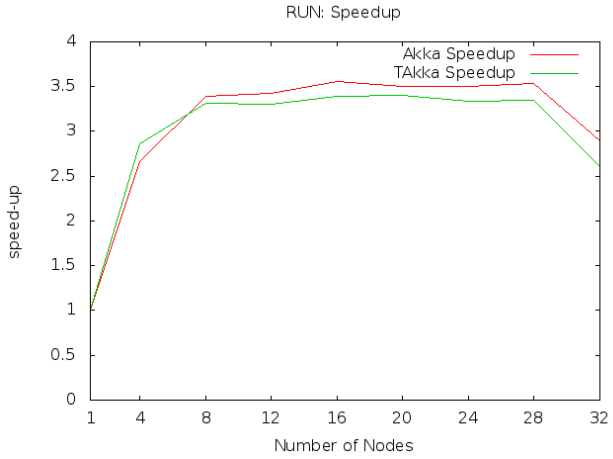
(b)



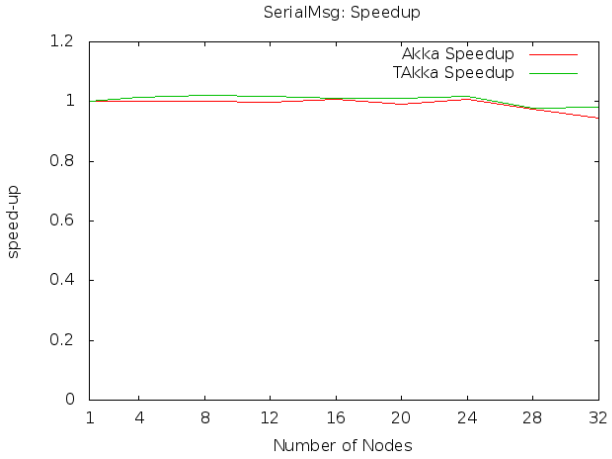
(c)



(d)



(e)



(f)

Figure 17. Scalability Benchmarks

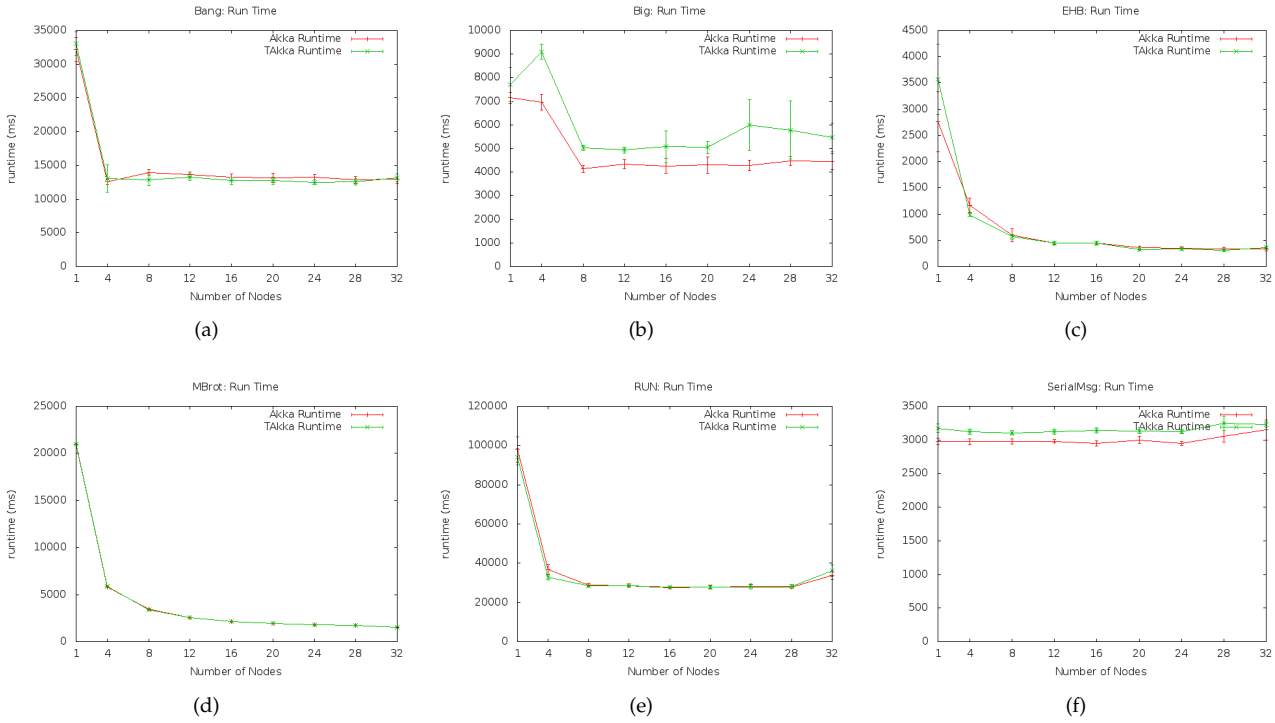


Figure 18. Runtime Benchmarks 2

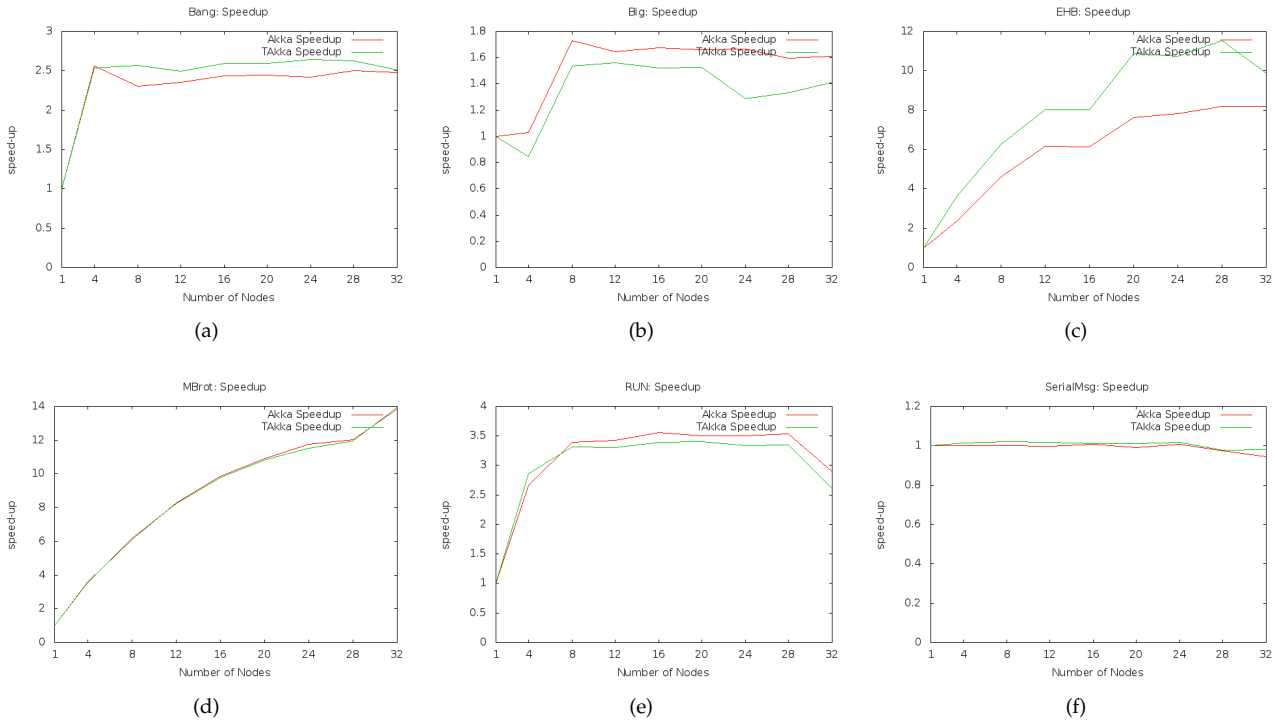


Figure 19. Scalability Benchmarks 2

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

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 circle because it takes the view master more than 5 seconds to receive their responses. From the result of this test, 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. As the test only runs for slightly more than 30 seconds, no responses to the last request are 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 time. In both cases, they are considered as alive in this test.

6.4.3 BenchErl Examples with Different Supervisor Strategies

To testing the behaviour of applications with internal states under different supervisor strategies, we apply the `OneForOne` supervisor strategy with different failure actions to the 8 BenchErl examples and test those examples using Chaos Monkey and Supervision View. The master node of each BenchErl tests 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 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 ChaosMonkey and Supervision View tests.

7. Conclusion

Existing actor systems 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 the compile time.

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

Lastly, tests show that building type-parameterized actors on top of Akka does not introduce significant overheads, with respect to program size, efficiency, and scalability. The above test results are encouraging in the sense that many previous actor library implementations can be re-used. We expect similar results can be obtained in other OTP-like libraries such as future extensions of CloudHaskell [Watson et al. 2012].

Acknowledgments

Acknowledgments

References

Jeff Fulmer . Siege Home. http://www.joedog.org/siege-home/, 2013. Accessed on July 2013.

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.

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.

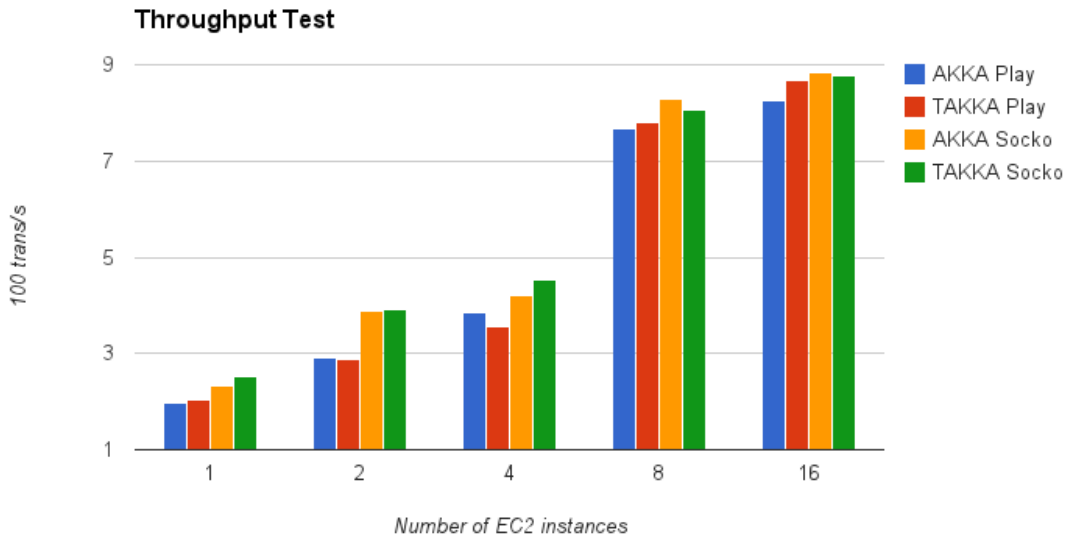
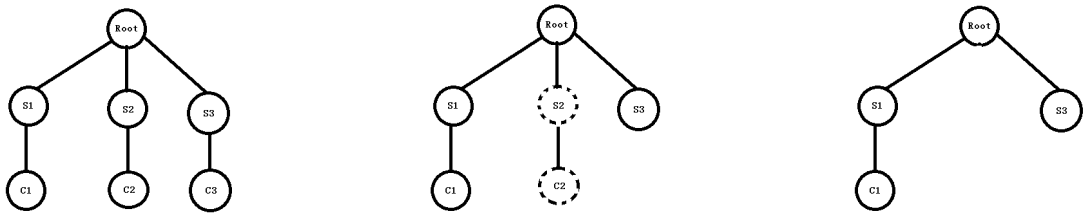


Figure 20. Throughput Benchmarks



(a) (b) (c)

Figure 21. Supervision View Example

Assessing the Reliability of Applications with Supervision Tree

Jiansen HE

1 Introduction

Libraries such as Erlang OTP [Ericsson AB., 2012], Akka[Typesafe Inc. (b), 2012], and TAkka are built with the belief that the reliability of a software application can be improved by using supervision tree, in which failed components will be restarted by their supervisor. Erlang OTP and Akka have been used in a number of applications that have achieved a high reliability. Could the reliability of a newly developed application be assessed in an effective manner? To what extend the usage of supervision tree contributes to the overall reliability?

To answer above questions, this proposal suggests a research roadmap as follows. Section 2 summarizes general methodologies used in the area of reliability studies. Focusing on the statistic approach to measure the reliability of software applications which are expected to have low failure rates, we propose to improve [Littlewood and Strigini, 1993]'s single-run experiment, summarized in Section 3.1, by using the iterative experiment (Section 3.2) when conditions apply. The iterative approach can reduce the experiment time and can be stopped when the desired belief about the reliability had been rejected. Aside from the methodology for directly assessing the reliability of a general application, this proposal also looks into approaches that indirectly assessing the reliability of application built using the supervision tree principle. Section 4 summarizes Nyström's approach for analysing the statical structure of Erlang supervision trees, and Jiansen's extensible work on dynamically monitoring TAkka supervision trees. Finally, the essence of a node in a supervision tree is abstracted as a Deterministic Finite Automaton

(DFA) in Section 5. The abstraction is used to model and compare designs of supervision tree in Erlang and (T)Akka, among alternatives that have not been adopted. A straightforward definition for the reliabilities of a node in a supervision tree is derived from the model. It remains to be seen whether i) the overall reliability of a supervising tree can be derived from the reliability of its nodes and the structure of that supervision tree, and ii) there exists an algorithm to aid the design of supervision tree with a high reliability.

2 Methodologies for Assessing Software Reliability

The reliability in this proposal is defined as the probability that a system can function properly for a specified period. Formally, the reliability function, $R(t)$, denotes the probability that a system can function properly for time t. In the study of software reliability, the period may be specified in term of *time units* (e.g. second) or *natural units* (e.g. number of operations) [Musa, 2004]. Reliability is usually reported in terms of failure rate (λ), Mean Time Between Failures (MTBF, $1/\lambda$), and other variants.

The reliability function, its failure density function $f(t)$, and MTBF has following relationships: [Musa, 2004]

$R(t) = \int_0^\infty f(x)dx$

$MTBF = \int_0^\infty tf(t)dt$

$MTBF = \int_0^\infty R(t)dt$

In literature, the precise definition of failures varies in the study of different systems. Failure in this proposal is a general term to describe the situation where a system or a sub-system does not work as specified.

Once the meaning of failure is defined for a specific application, measuring the reliability during a period is straightforward. The challenge is how to use the previous data, collected from experimental or operational environment, to give confidence about the reliability in the future under the operational environment.

For software which involves iterative debugging process, software reliability growth models are usually employed to capture the relationship between the bug removing process and the reliability improvement. [Abdel-Ghaly et al., 1986] examines 10 models in this area, and a framework to evaluate the effectiveness of different models.

For software that has reached certain level of reliability and hence the debugging process has minor effects, [Littlewood and Strigini, 1993] gives a Bayesian approach to validating its reliability. Because this approach targets at systems that have high reliabilities, we will discuss this approach in sufficient detail in the next section.

3 Statistic Approaches for Measuring Reliability

3.1 Littlewood's approach to predict software reliability.

The analysis in [Littlewood and Strigini, 1993] answers the following question: given that x failures are observed during the period of testing t_0, what can we conclude about the reliability in the next t time.

The two assumptions in Littlewood's approach are:

Assumption I) the occurrence of failures is a poisson process, that is, time between failures are independent.

Assumption II) the prior belief about the distribution of failure rate, $p(\lambda)$ follows the gamma distribution, $\text{Gam}(\alpha, \beta)$, where α and β are positive hype-parameter that describes the sharp and the rate of the distribution.

Assumption I is an appropriate choice for general studies.

Since the posterior belief is adjusted by the observed data, the prior belief will have less effect on the posterior belief if sufficient evidence is collected from a long experiment. The Gamma distribution is chosen because it is the *conjugate distribution* for poisson distribution. It means that using the Gamma distribution as prior belief will result to a posterior belief which also follows the Gamma distribution. Particularly, if the prior belief is $\text{Gam}(a, b)$, then posterior belief will be $\text{Gam}(a + x, b + t_0)$, where x is the number of observed failures and t_0 is the experiment time. Littlewood and Strigini Alternatively, any probability distribution may be used to replace the Gamma distribution, if it better describes the property of the failure rate of a particular application.

Littlewood and Strigini then derives that the reliability function for the future t time is

$R(t|x, t_0) = \left(\frac{b+t_0}{b+t_0+t}\right)^{a+x}$

Littlewood and Strigini then concludes that "observing a long period of failure-free working does not in itself allow us to conclude that a system in ultra-reliable" from the analysis of two extreme cases. The first case is, choosing an improper prior so that the posterior completely depends on the data. The posterior itself is an proper distribution; however, after observing a period of failure-free operations, one has 50% possibility to be failure-free for the same amount of time in the future. The second example is, to claim a system has 10^6 hours (114 years) MTBF by showing 10^3 hours (41.7 days) of failure-free working, one must hold the prior belief that the system is a 10^6 system.

3.2 Reduce experiment time of the Littlewood and Strigini test?

In Littlewood and Strigini's study, prediction of the reliability is made according to the result of an earlier experiment under the same operational environment. In such an experiment, even a long-time failure free observation cannot conclude a high reliability for the future. This section attempts to solve three problems: i) based on the same assumptions about the failure rate, can we improve the experiment so that the expected reliability can be verified or rejected earlier? ii) can we validate the same basic assumptions used in Littlewood and Strigini's and the improved approach? iii) is it possible to claim a genuine prior belief about the distribution of the failure rate?

3.2.1 Iterative experiments

Assuming that the occurrence of failures is a poission process and the failure rate follows the Gamma distribution, representing the Gamma distribution using hyper-parameters, it is clear that the posterior belief about the failure rate, $Gam(a + x, b + t)$, depends on the prior belief $Gam(a, b)$, the *total* number of observed failures x, and the *total* experiment time t. Therefore, for an application that consists of a number of replica subsystems, we can replace Littlewood and Strigini's experiment with an equivalent iterative experiment described below.

An iterative experiment consists of several rounds of sub-experiments, each of which may consists of several parallel experiments. In an iterative experiment, the posterior belief of one round is used as the prior belief of the next round. Instead of asking whether a system will be reliable in the next 10^6 hours based on the failure rate of the first 10^3 hours [Littlewood and Strigini, 1993], the experiment conductor concerns whether the system will be reliable in the next round, based on the failure rate of previous results. In each iteration, a number of instances may be tested in parallel to collect the results of more instance hours within less time. An iterative experiment may be stopped as soon as the belief about the failure rate is disproved, or the base assumption about poisson process and Gamma distribution are rejected.

Take the above proposal into practise, assuming we are asked to test whether a distributed web service designed for an organisation will be as reliable as its developers claimed for the next a few years. We may run a test on one or two local machines for 1 day. If the result meets the requirement, then we may test the application on some servers of the organisation for half a week. Then, we may rent 1000 similar servers from one or more cloud service providers in the third round which lasts a week. By doing this, we can collect the number of failures occurred in an experiment of 168,000 instance hours within a week ($168,000 = 7 \times 24 \times 100$) . Fortunately, the data can be treated as equivalent to a result collected from a single-run experiment on one machine for 19 years ($19 = 168,000 \div 24 \div 365$) , if our assumptions are valid for tested system. Renting a thousand server instances might be expensive, but the risk of wasting resource on testing application that doesn't meet the desired reliability has been reduced in previous rounds of tests.

3.2.2 Verify Assumptions

Both Littlewood and Strigini's original approach and our alternative are based on the assumption that the occurrence of failures is a poisson process and the failure rate follows a Gamma distribution represented by two hyper-parameters. Those two assumings are good choice in a study for general purposes. For a test of a real application, to what extend can we rely on the assumption of poisson process and the prior belief with guessed hyper-parameters? To verify those assumptions, suitable goodness-of-fit tests can be used. Following are example approaches.

To verify the assumption on poission process, we can alternatively check

whether the number of failures in consecutive fixed periods follows the poission distribution, whose mean and variance are the same. To verify the assumption on gamma distribution, [Woodruff et al., 1984] gives improved methodologies for the purpose of reliability test.

3.2.3 Obtain a genuine prior belief

Following is my guesswork. Appropriate statistic analysis is required.

If x failures are observed in the previous experiment of t time and the base assumption cannot be rejected by the data in previous tests, the **best** prior belief for the next iteration is assuming the distribution of failure rate λ follows $\mathrm{Gam}(x,\, t)$.

This is a significant difference between the iterative experiment and Littlewood and Strigini's single run experiment. In Littlewood and Strigini's approach, claiming a $\mathrm{Gam}(x,\, t)$ *posterior* is the same as claiming a improper *prior* $\mathrm{Gam}(a,\, b)$, where $a,\ b\ \to 0$. In an iterative experiment, the first round can be used to "generate" a reliable prior for the second round. However, at least 1 failure need to be observed in the first round because a and b in Gamma distribution are positive numbers. In a ultra-reliable system, waiting for the first occurrence of failure may take a long time.

3.2.4 Testing in adversely environment.

Another attempt to give a reliability prediction in short period is testing the system in adversely environment. This idea, known as accelerated testing (AT), has been explored in hardware reliability test for years [Escobar and Meeker, 2006], and has been applied to test the $FASTAR^{SM}$ platform, a set of systems used to automatically restore the AT&T network[Cukic, 1997].

An accelerated test consists of two general steps. The first step is to identify accelerating factors (e.g. temperature, workload, temperature etc.) and run the experiment. The second step is to use suitable regression model to predict the reliability in normal condition.

The significance of an accelerated testing depends on the choice of the regression model. For a hardware, the relationship between its life-time and environment factors (e.g. temperature) has been well studied. In the $FASTAR^{SM}$ test, to capture and verify the regression model, 32 runs are carried out under different pressure environment. Each run lasts for about 30 hours and a few failures are observed during the test.

Bring in the same idea into the test of TAkka applications, for example, with the help of the Chaos Monkey library, can we effectively test the reliability within a reasonable short period? Unfortunately not.

Let's run two experiments in parallel. In one experiment, ChaosMonkey is not used and its failure rate, λ_n, is assumed to be the same as in the normal operational environment. In the other experiment, ChaosMonkey is employed to increase the exception rate so that the failure rate, λ_c, will be higher than λ_n. Failure rates in the two experiments are linked by the following equation: $\lambda_n \times t_n = \lambda_c \times t_c$, where t_n and t_c are expect periods of seeing x failures in the normal environment and in the ChaosMonkey test.

The above equation shows that the expected time for the nth occurrence of failures decreases when its failure rate increased. Therefore, assumptions on the poission process and the Gamma distribution can be verified in shorter time. From a serials of ChaosMonkey test with different failure rates, we may fortunately find that the occurrence of failures in the tested application is always a poission process. As a result, to measure its reliability in normal operational environment, only a small number of errors need to be observed under the test in normal environment. However, it may take a long time to observe the first occurrence of failures in a system whose MTBF is 10^6 hours.

4 Reliability Studies on Supervision Tree

4.1 Static Analysis on Tree Structure

Based on the Core Erlang Language defined by [Carlsson et al., 2000], [Nyström, 2009] defines an abstract semantic to extract structures of Erlang programs. Since supervision tree is constructed in Erlang by calling callback functions in the *supervision* module, Nyström's work can be used to abstract the structure of supervision trees form the source code of Erlang applications.

Noted in the later part of Nyström's thesis, there are two limitations of his approach. Firstly, the abstraction only captures the static structure of the supervision tree, which may differ from its run-time structure. Secondly this approach requires specialised knowledge about the Erlang language and only applies to "applications that have been designed according to the suggestion of the OTP documentation and using the OTP library behavious." [Nyström, 2009].

4.2 Dynamic Monitoring

The supervision view library in the TAkka paper.

5 Modelling Supervision Tree

To study properties of supervision tree, a simple formal model is required to describe the essence of supervision tree. Section 5.1 models Worker and Supervisor and Deterministic Finite Automata (DFA). With this model, implementation alternatives of Supervision Tree are compared. More importantly, the model helps us to give a simple general definition for the reliability of nodes in a supervision tree. This section ends with a proposal for investigating the reliability of a supervision tree, based on the reliabilities of its nodes and the relationships between nodes.

5.1 Worker and Supervisor as Deterministic Finite Automata (DFA)

Figure 1 gives state graphs of DFAs that describe worker and supervisor in a supervision tree. Nodes in the graphs represent states of that DFA and arrows are transitions from one state to another. A transition is triggered either by the node itself or due to changes of the environment it resides.

At any time, a worker may be in one of its four states: **Start**, **Free**, **Blocked**, or **Dead**. After automatically being initialised to the **Free** state, the Work can accept messages from its outside environment and enters to the **Blocked** state where no messages will be processed. If no error occurred when processing the message, the worker emits a result and go back to the **Free** state. An error may occur during the middle of message processing (e.g. software bugs) or when the environment changes (e.g. hardware failures), in which cases the worker reports its failure to its supervisor and go to the **Dead** state. A dead worker can be resumed or restarted by its supervisor so that it can process new messages. **Free** and **Dead** are marked as accept states from which no further actions may occur. On the contrary, a **Blocked** worker will eventually emit a reply message or raise an exception; and all workers will be successfully initialised.

The DFA of a supervisor whose only duty is supervising its children is given in Figure 1(b). A supervisor in its **Free** state reacts to failure messages

from its children. Meanwhile, a supervisor may fail at any time and reports its failure to its supervisor.

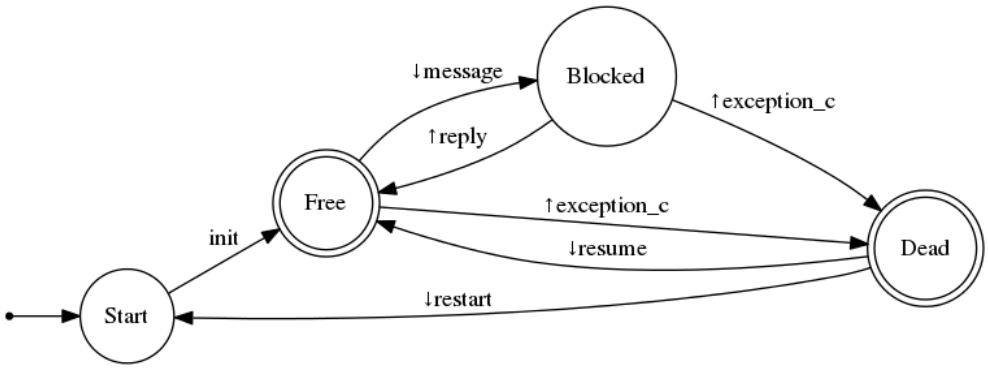
5.2 Implementation Considerations

The two DFAs given in Figure 1 are abstracted for theoretical study. In an implementation of the supervision tree model, following issues might be considered.

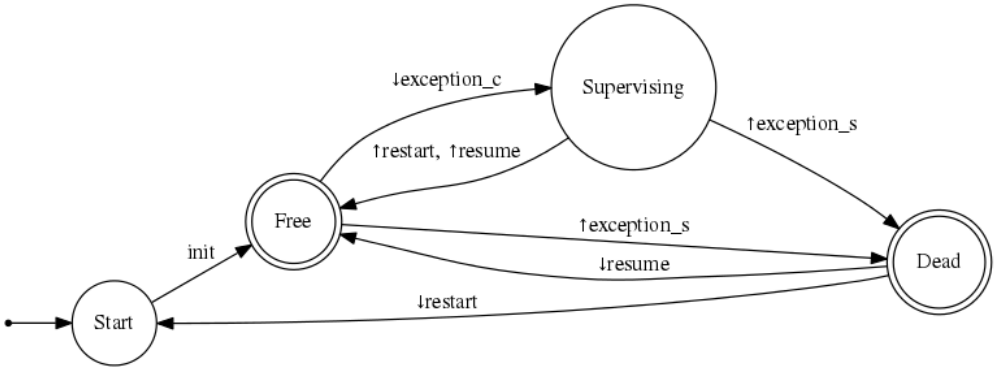
Distributed Deployment Supervision Tree represents the logic relationship between nodes. In practice, nodes of a supervision tree may be deployed in distributed machines. A child node may be restarted at or shipped to another physical or virtual machine but stays in the same place in the logic supervision tree.

Heart-Beat Message At run-time, failures may occur at any time for different reasons. In some circumstances, failure messages of a child may not be delivered to its supervisor, or even worse, not sent by the failed child at all. To built a system tolerant to the above failures, supervisors need to be aware of the liveness of their children. One approach to achieve this is asking the child periodically sending heart-beat message to its supervisor. If no heart-beat message is received from a child for some consecutive periods, that child is considered dead by the supervisor and an appropriate recovery process is activated. In the model, a logical exception is sent from a child to its supervisor when no heart-beat message is delivered within a time-out, and a restart or resume message is sent to a suitable machine where the node will reside.

Message Queuing In the simplified model given in Figure 1(a), a worker can processed one message a time when it is **Free**. The model does not exclude the case where messages are queuing either in memory or a distributed database. Similarly, when a worker is resumed from the failure of processing a message, the message it was processing may be retrieved from a cache or be discarded.



(a) Worker



(b) Supervisor

Figure 1: Worker and Supervisor as Deterministic Finite Automata

5.3 Unifying Supervisor and Worker

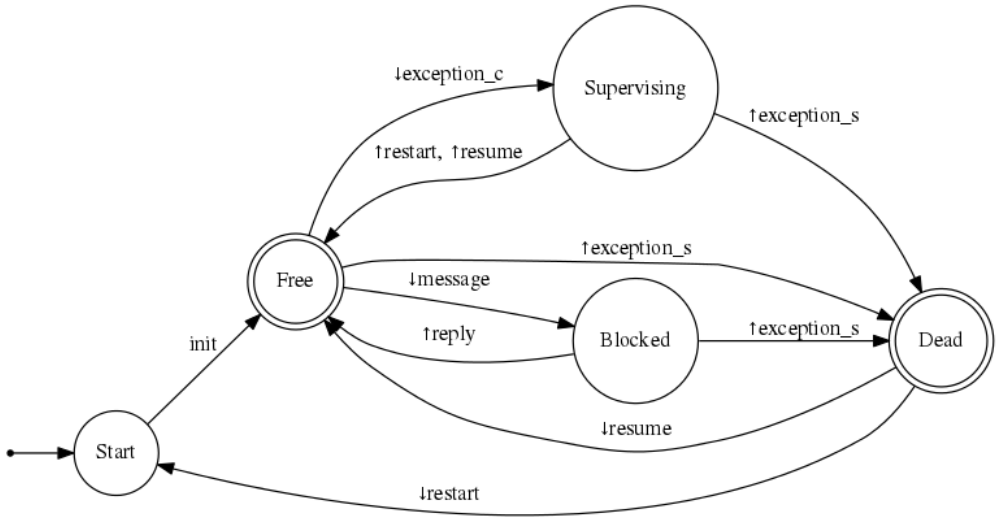
Notice that the model for supervisor and worker are similar if exceptions from the child is viewed as request messages to a supervisor and restart/resume is viewed as reply messages to a child. Can we defined a *combined* node which can be both a worker for a task and a supervisor for some children? Of course we can, as what have been implemented in Akka[Typesafe Inc. (b), 2012] and inherited by TAkka. The rest of this sub-section will compare three strategies of unifying supervisor and worker.

Supervisor as a Worker In the design of Akka, messages to actors are not typed. An Akka actor therefore can be both a supervisor and a worker. To model an Akka actor, only Figure 1(a) is required.

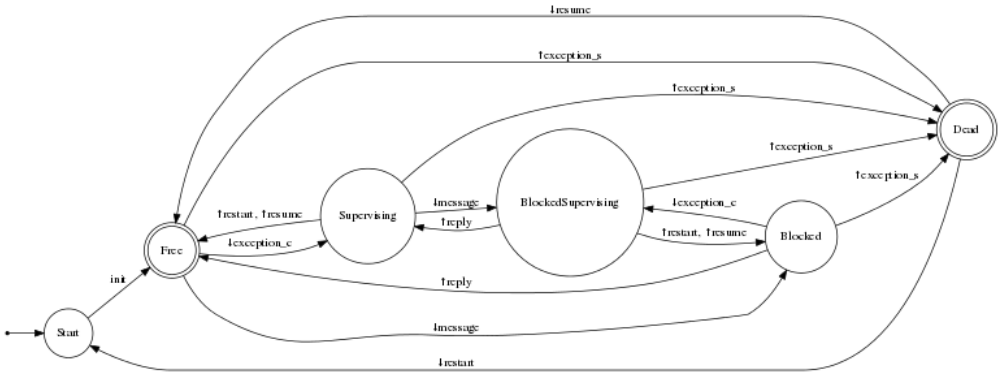
Combining Supervisor and Worker The TAkka library inherits the implementation of Akka, but separates messages for supervision purpose from messages for general purposes so that an actor can be parameterized by the type of message it expects. A more precise model for a TAkka actor is given in Figure 2(a). Both Akka actor and TAkka actor can only be a supervisor or a worker at a time. As a result, the supervision task may be blocked until the end of another computational task.

Supervisor in Parallel with Worker One way to get around the limitation of Akka and TAkka's design is place the supervision process and the worker process in parallel, as shown in Figure 2(b). From the perspective of model analysis, the above parallel model is equivalent to the one which separates the supervisor process and the worker process into two nodes, and treat them as siblings or a supervisor and a child.

To summarize, a node that naively combining the role of supervisor and worker (Figure 1(a) and 2(a)) has less availability than a node that place the supervision process and the worker process in parallel processes (Figure 2(b)). Interestingly, during the process of designing TAkka, we realized that the type of supervision messages should be separated from the type of other messages. However, partly because we would like to reuse the Akka implementation and partly because we did not have the above model at that time, the process of handling supervision messages is not separated from the process of handling other messages.



(a) Supervisor AND Worker



(b) Supervisor PAR Worker

Figure 2: A Node that Unifies Supervisor and Worker

5.4 Reliability of a Node

The reliability of a node, either a worker or a supervisor, is defined as:

The the probability that a node is in the **Free** state.

5.5 Reliability of a Supervision Tree

A supervision tree in this proposal consists of workers and supervisors defined in Figure 1(a) and Figure 1(b) respectively. A node that combines a supervisor and a worker is separated into two nodes, together with a constraint that one node will fail when the other fails.

The reliability of a supervision tree may be derived from following factors:

- The reliabilities of all workers. Although testing the reliability of a system with low failure rate is difficult or even impractical, the reliability of an individual component may be measured within a reasonable short time.
- The reliability of a supervisor. The reliability of a supervisor process may be tested as part of the library development process.
- The relationship between nodes. Nodes in a supervision tree may collaborate to perform a task or to achieve a higher reliability. The reliability of a supervising shall capture complex relationships between nodes.

The experiment proposed in section 3.2 may to used to measure the reliabilities of individual nodes in a supervision tree. To obtain the reliability of a supervision tree, when direct measurement becomes impractical, knowledge about constraints between nodes are required.

I propose to investigate following problems in the next stage:

- What are possible constraints between nodes? For each constraint, what is the algebraic relationship between the reliability of a sub-tree and reliabilities of individual nodes?
- Based on the above result, how to calculate the overall reliabilities of a supervision tree? When is the reliability is improved by using supervising tree, and when not?

- Given the reliabilities of individual workers and constraints between them, is there an algorithm to give a supervision tree that improves the reliability? If not, can we determine if the desired reliability is not achievable?

References

A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood. Evaluation of competing software reliability predictions. *IEEE Transactions on Software Engineering*, 12(9):950–967, 1986. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.1986.6313050.

R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000-030, Department of Information Technology, Uppsala University, Nov. 2000.

B. Cukic. Accelerated testing for software reliability assessment, 1997.

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

L. A. Escobar and W. Q. Meeker. A review of accelerated test models, 2006.

B. Littlewood and L. Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36:69–80, 1993.

J. D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper 2nd Edition*. 2 edition, Sept. 2004. ISBN 9781418493882.

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

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.

B. W. Woodruff, P. J. Viviano, A. H. Moore, and E. J. Dunne. Modified goodness-of-fit tests for gamma distributions with unknown location and scale parameters. *Transactions on Reliability*, R-33:241 – 245, 1984.