Distributed Programming with Types and OTP Design Principles: Second Year Ph.D Progress Report

Jiansen HE

October 25, 2012

Abstract

This report will first give a review on project motivation and objectives, followed by an overview of current progress. This document will also elaborate a framework for library evaluation, which is developed after the last quarterly review.

1 Project Motivation

In 1996, the Erlang/OTP (Open Telecom Platform) library[1] was released for writing Erlang code using the Actor model[9] and principles derived from 10 years successful experience. Those principles are later known as OTP principles. Although Erlang is an untyped language, with the help of testing tools and OTP principles, successful OTP applications have achieved a high reliability.[2]

Since May 2011, the Akka team has been working on porting OTP principles into Scala, a typed language that mixes functional and objected-oriented programming paradigms. The Akka library[12, 11] demonstrated the possibility of using OTP principles in a typed setting; however, messages to Akka actors are dynamically typed and some Akka APIs may deserve more specific types. It remains to be seen to what extent a strongly typed platform helps the construction of reliable distributed applications.

2 Project Objectives

The aim of this project is to *study how types and OTP principles help the construction of distributed systems.* This project intends to explore factors that encourage programmers to employ types and good programming principles. To this end, following iterative and incremental tasks are required.

1. To identify some applications implemented in Erlang or Akka using OTP 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, (d) eventual consistency model for shared states, and (e) dynamic topology and hot code swapping.

2. To implement a library, TAkka, that supports actor programming and OTP design principles in a strongly typed setting. Basic components of the library may be built on top of Akka[12] 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.
3. To re-implement identified applications using the library developed in task 2. The primitive purpose of this task is to be aware of prominent features and limitations of existing and the proposed library. 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 OTP libraries meet the demands of distributed programming. This project suggests evaluating OTP libraries respecting to efficiency, scalability, and reliability. To give an overall evaluation regarding to several dimensions, a novel methodology is required. The evaluation phase does not suggest the end of this project. The library and its evaluation will be published to programming communities for feedbacks before we conclude.

3 Progress Overview

At the time of this writing, key components of the TAkka library have been implemented. The TAkka library provides type-parametrized actor and Akka-style APIs. The correctness and usability of TAkka library has been tested by porting Erlang and Akka examples. Experience shows that type-parametrized actor works well with OTP principles and provides a clearer communication interface and better guarantee of type safety.

Aside from type-parametrised actor, I have implemented a typed nameserver which maps each type symbol to a value of corresponding type. The typed nameserver is used in the TAkka library for looking up typed actor references.

In addition, I demonstrated how to solve the type pollution problem using subtyping. Without due care, the type pollution problem may occur when a service or communication channel is shared by two distinct parties, which is a common scenario in systems constructed using layered architecture or the MVC model. Layered architecture is a common model for building web services while MVC model is the de facto model for building mobile applications on Android and iOS platforms. Therefore, principle proposed in this piece of work may impact on the design of distributed systems in more general settings.

In the last 4 months, a framework for evaluating OTP libraries has been developed. The evaluation methodology will be elaborated in the next part.

4 Methodology for Library Evaluation

The three dimensions of library evaluation are (i)efficiency, (ii)scalability, and (iii)reliability. The overall evaluation of the library is the geometric mean of numerical measurements of each dimension. Standard numerical measurements for evaluating efficiency has been found in literature. A similar approach may apply to the evaluation of scalability. A numerical measure of reliability, unfortunately, requires further study. The first section of this part will review methodology of hardware efficiency evaluation, which I believe applies to software efficiency evaluation as well. In the rest three sections of this part, approaches to evaluate each dimension will be examined respectively.

4.1 Measuring Hardware Efficiency

In practice, hardware efficiency is measured by its performance regarding to a benchmark suite, which contains a set of test programs. Performance is defined as the reciprocal of mean execution time of all tests.[8] This project will use geometric mean instead of weighted arithmetic mean for three reasons.

Firstly and most importantly, as proved by Fleming and Wallace [6], geometric mean is the (only) correct average of normalised measurements. As users often compare the performance of one system to another, performance is usually normalised to a reference system. Precisely, let the benchmark suite has n tests indexed from 1 to n, performance of system A is calculated as:

$$performance_A = \frac{time_{ref}}{time_A} = \frac{\sqrt[n]{\prod_{i=1}^{n} time_{ref_i}}}{\sqrt[n]{\prod_{i=1}^{n} time_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{time_{ref_i}}{time_{A_i}}}$$

where $time_{S_i}$ is the execution time of program i on system S, and $time_S$ is the time measurement of system S.

Secondly, as shown in the above equation, the order of calculating means and normalisation is flexible. For the above reason, the suite developer will only implement a generic tool for measuring unnormalised execution time. The work of comparing performance of two interested platforms is left to users. In addition, users are free to extend the benchmark suite. When new tests are added to the benchmark suite, performance measure for old examples becomes a valid partial measure.

Finally, statistic tools could be employed to verify the soundness of chosen suite. For example, the geometric standard deviations is an indicator to assess the variability of the chosen suite.

4.2 Measuring Library Efficiency

The Methodology of measuring hardware efficiency also applies to measuring library efficiency, where both the library implementation and the running platform may differ.

To test the efficiency of different distributed programming libraries, a set of benchmarking examples need to be identified. To avoid unintentional optimisation in the TAkka implementation, benchmarks are scrupulously selected from existing applications written in other OTP implementations and modifications are made at the minimal level.

Theoretically, any program that involves a time measurement could be a candidate efficiency test. Such examples could be easily found in both the correctness test suite and the scalability test suite. However, we would like to exclude examples whose results significantly determine the overall measure. To this end, statistic methods such as principal component analysis will be employed.

4.3 Measuring Scalability

To assess the scalability of different OTP implementations, suitable examples from the BenchErl suite[1][4] have been reimplemented using Akka and TAkka. A summary of selected examples will be given at §6.

Similar to efficiency evaluation, geometric mean of scalability measures of all examples could be the indicator of the library scalability. Unfortunately, numerical evaluation for scalability is not found in the literature. This project will use linear regression techniques to analysis the scalability of an OTP library in two aspects: (i) the degrees to which the performance speedup deviate from the ideal linear relationship regarding to the number of active CPU cores. (ii) the slope of the linear equation, assuming that the relationship between performance speedup and the number of active CPU cores is linear.

Finally, scalability of OTP implementations will be tested on a variety of platforms. The three candidate platforms are Beowulf cluster, Google App Engine, and Amazon EC2.

4.4 Measuring Reliability

Identify types of failures and their impact is a crucial part of reliability evaluation. The next a few months will be devoted to the study of possible failures that could be handled at the language level. Methodology for numerically evaluating reliability requires more studies and careful verifications.

[1]BenchErl is a scalability benchmark suite for Erlang/OTP. It is developed as part of the RELEASE project.

5 Examples for Correctness and Usability Test

This section lists examples used for correctness and usability test. Most of those examples are ported from other projects so that main requirements for OTP programming will not be neglected. For each example, the number of modified lines and the total number of lines in the Akka version are given in the brackets.

5.1 Examples Ported from Quviq Training Examples

Following examples are ported from examples in a Quviq course[2][3]. To avoid unintentional leakage of their Erlang design, Akka and TAkka implementation for those examples are not published to public repositories.

- ATM simulator (199/1148): A bank ATM simulator with backend database and frontend GUI.
- Elevator Controller (172/1850): A system that monitors and schedules a number of elevators.

5.2 Examples from Akka documentation

Following examples are selected from Akka documentation[12] to check that the TAkka library could provide Akka equivalent services. Those examples are:

- Ping Pong (13/67): A simple message passing application.
- Dining Philosophers (23/189): A application that simulates the dining philosophers problem using Finite State Machine (FSM) model.
- Distributed Calculator (43/250): An application that examines distributed computation and hot code swap.
- Fault Tolerance (69/274): An application that demonstrates how system responses to component failures.

5.3 Other Examples from Open Source Akka Applications

Following examples are selected to check that the TAkka library meets the demand of building real concurrent applications. Those examples are: (ref for all below)

- Barber Shop[13] (104 / 754): The implementation of famous Barber Shop example in Akka.

[2]Quviq is a quickcheck tool for Erlang programs

- EnMAS [5] (213/1916): An environment and simulation framework for multi-agent and team-based artificial intelligence research
- Socko Web Server [10]: “ lightweight Scala web server that can serve static files and support RESTful APIs to our business logic implemented in Akka.”
- Gatling [7]: A stress testing tool. Most Gatling components are implemented using its own DSL. Therefore, changes are only made to 27 core files, where 111 out of 1635 lines of code are modified.

6 Examples for Scalability Test

Following 9 examples are ported from the BenchErl suite[4] for the purpose of scalability test.

- 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.
- parallel: This benchmark spawns a number of processes, where a list of N timestamps is concurrently created.
- genstress: This benchmark is similar to the bang test. It spawns an echo server and a number of clients. Each client sends some dummy messages to the server and waits for its response. The Erlang version of this test can be executed with or without using the gen_server behaviour. For generality, this benchmark only tests the version without using gen_server.
- serialmsg: This benchmark tests message forwarding through a dispatcher.
- timer_wheel: This benchmark is a modification to the big test. While responding to ping messages, a process in this message also waits pong messages. If no pong message is received within the specified timeout, the process terminates itself.
- 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.

The other 4 tests in the BenchErl suite are not selected because they are testing specific features of the Erlang platform.

7 Plan

Following tasks will be carried out in the next year.

Month	Task
Nov 2012	Test scalability on different platforms
Nov 2012 - Dec 2012	Define failure model Build benchmark suite for reliability testing Define methodology for numerically evaluating reliability
Jan 2013 - Feb 2013	Work on large examples such as Riak and Play2
Mar 2013	Improve the takka library
Apr 2013	Start writing the thesis Study recent relevant research Design the routine of future work

References

[1] Ericsson AB. Erlang Programming Language, 2012.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf 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.

[4] 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.

[5] Connor Doyle and Martin Allen. Release: a high-level paradigm for reliable large-scale server software. *Midwest Instruction and Computing Symposium*, 2012.

[6] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[7] Excilys Group. Gatling: stress tool, June 2012.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, September 2006.

[9] Carl Hewitt, Peter Bishop, and Richard 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.

[10] Vibul Imtarnasan and Dave Bolton. Socko Web Server, June 2012.

[11] Typesafe Inc. Akka API: Release 2.0.2, April 2012.

[12] Typesafe Inc. Akka Documentation: Release 2.0.2, June 2012.

[13] Martin Zachrison. Barbershop, 2010.

A The TAkka Library

This appendix gives an introduction to key features in the TAkka library. Full Takka APIs could be found at `http://homepages.inf.ed.ac.uk/s1024484/takka/snapshot/api/`.

A.1 Type Parametrized Actor

Every TAkka actor takes a type parameter specifying the type of its expecting messages. Only messages of the right type can be sent to TAkka actors. Therefore, receivers do not need to worry about unexpected messages while senders can ensure that messages will be understood and processed, as long as the message is delivered.

Table-1 shows how to define and use a simple string processing actor in Akka and TAkka. Both actors define a message handler that only intends to process strings. At line 15, sending a message of non-string type to the string processing actor is prohibited in TAkka but allowed in Akka[3].

```
1  import akka.actor._
2
3  class MyAkkaActor extends Actor{
4    def receive = {
5      case m:String =>
6       println("received message: "+m)
7    }
8  }
9
10 object AkkaActorTest extends App {
11   val system = ActorSystem("MySystem")
12   val myActor = system.actorOf(
13          Props(new MyAkkaActor),"myactor")
14   myActor ! "hello"
15   myActor ! 3
16  //no compile error
17
18 }
```

```
1  import takka.actor._
2
3  class MyTAkkaActor extends Actor[String] {
4    def typedReceive = {
5      case m =>
6        println("received message: "+m)
7    }
8  }
9
10 object TAkkaActorTest extends App {
11   val system = ActorSystem("MySystem")
12   val myActor = system.actorOf(
13          Props[String](new MyTAkkaActor),"myactor")
14   myActor ! "hello"
15   //myActor ! 3
16   //compile error: type mismatch; found : Int(3)
17   // required: String
18 }
```

Table 1: A Simple Actor in Akka and TAkka

A.2 Supervising Type Parametrized Actors

System reliability will be improved if failed components are restarted in time. The OTP Supervision Principle suggests that actors should

[3]In versions prior to Akka 1.3.1, receiving a message that does not match any pattern will cause an exception. Since Akka 2.0, unhandled messages are sent to an event stream. Both strategies are different from the Erlang design where unhandled messages are remaining in the actor's message box for later processing.

be organised in a tree structure so that failed actors could be properly restarted by its supervisor. The Akka library enforces all user-created actors to be supervised another actor. The TAkka library inherits the above good design. To be embedded into a supervision tree, every TAkka actor, Actor[M], defines two message handlers: one for messages of type M and another for messages for supervision purpose.

A.3 Wadler's Type Pollution Problem

The Walder's type pollution problem is first found in layered systems, where each layer provides services to the layer immediately above it, and implements those services using services in the layer immediately below it. If a layer is implemented using an actor, then the actor will receive both messages form the layer above and the layer below via its only channel. This means that the higher layer could send a message that is not expected from it, so as the lower layer.

Generally speaking, type pollution may occur when an actor expects messages of different types from different users. Another example of type pollution is implementing the controller in the MVC model using an actor.

One solution to the type pollution problem is using type parametrized actors and sub-typing. Let a layer has type Actor[T], denoting that it is an actor that expects message of type T. With sub-typing, one could publish the layer as Actor[A] to one user (e.g. the layer above or the Model) and publish the same layer as Actor[B] to another user (e.g. the layer below or the Viewer) at the same time. The system could check if both A and B are subtypes of T.

A.4 Code Evolution

Same as in the Akka design, an actor could hot swap its message handler by calling the *becomes* method of its context, the actor's view of its outside world. Different from the Akka design, code evolution in TAkka has a restricted direction, that is, an actor must evolve to a version that is able to handle the same amount of or more patterns. The decision is made so that a service published to users would not become unavailable later. The related code in the ActorContext trait in presented below. Notice that both static and dynamic type checking are used in the implementation.

A.5 A comparison with 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 typed actor invoke a service by calling a method instead of sending messages. The typed actor prevents some type errors but has some limitations. For one thing, typed actor does not permit code evolution. For the other, typed actor cannot solve the type pollution problem.

```scala
trait ActorContext[M] {
 implicit var mt:Manifest[_] = manifest[M]

 def become[SupM >: M](behavior: SupM => Unit,
     possibleHamfulHandler:akka.actor.PossiblyHarmful =>
     Unit)(implicit smt:Manifest[SupM]):ActorRef[SupM] = {
     if (!(smt >:> mt))
       throw BehaviorUpdateException(smt, mt)

     mt = smt
     // other code
 }
}
```

Table 2: Code Evolution in TAkka

A.6 Typed Nameserver

Traditional nameserver associate each name with a value. Users of a traditional name server can hardly tell the type of request resource. To overcome the above limitation, we designed a typed nameserver which map each typed name to a value of the corresponding type, and to look up a value by giving a typed name.

A.6.1 Scala Manifest as Type Descriptor

Comparing type information among distributed components requires a notion of type descriptor that is a first class value. In Scala, a descriptor for type T could be constructed as manifest[T], which is a value of type Manifest[T]. Furthermore, with *implicit parameter* (e.g. line 4 below) and *context bound* (line 10 below), library designer could present users concise APIs. For example, in a Scala interpreter, one could obtain the following results:

```
1  scala> manifest[Int => String]
2  res0: Manifest[Int => String] = scala.Function1[Int,
       java.lang.String]
3
4  scala> def name[T](implicit m:
       scala.reflect.Manifest[T]):String = m.toString
5  name: [T](implicit m: scala.reflect.Manifest[T])String
6
7  scala> name[Int => String]
8  res1: String = scala.Function1[Int, java.lang.String]
9
10 scala> def name2[T:Manifest]:String =
       {manifest[T].toString}
11 name2: [T](implicit evidence$1: Manifest[T])String
```

```

scala> name2[Int => String]
res2: String = scala.Function1[Int, java.lang.String]
```

A.6.2 Typed Name and Typed NameServer

A typed name, TSymbol, is a unique string symbol shipped with a manifest. In TAkka, TSymbol is defined as follows:

```
case class TSymbol[T](val symbol:Symbol)(implicit val
    t:Manifest[T]) {
  override def hashCode():Int = symbol.hashCode() }
```

The three operations provided by typed nameserver are:

- set[T:Manifest](name:TSymbol[T], value:T):Unit
  Register a typed name with a value of corresponding type. A 'NamesHasBeenRegisteredException' will be thrown if the name has been used by the name server.
- unset[T](name:TSymbol[T]):Unit
  Cancel the entry *name* if (i) the symbol representation of *name* is registered, and (ii) the manifest of *name* (i.e. the intention type) is a super type of the registered type.
- get[T] (name: TSymbol[T]): Option[T]
  Return Some(v:T) if (i) *name* is associated with a value and (ii) T is a supertype of the registered type; otherwise return None.

Notice that *unset* and *get* operations permit polymorphism. For this reason, the hashcode of TSymbol does not take manifest into account. The typed nameserver also prevents users from registering two names with the same symbol but different manifests, in which case if one is a supertype of another, the return value of *get* will be non-deterministic.