

Reliable Distributed Programming via Type-parameterized Actors and Supervision Tree

Jiansen HE

Master of Philosophy
Department of Computer Science
University of Edinburgh
2014

To ...

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(Jansen HE)

Table of Contents

Chapter 1	Introduction	1
1.1	Contributions	1
Chapter 2	Background and Related Work	3
2.1	Actor Programming	3
2.2	The Supervision Principle	3
2.3	The Erlang Programming Language	4
2.4	The Akka Library	13
Chapter 3	TAkka: Design and Implementation	22
3.1	Mixing Static and Dynamic Type Checking	22
3.2	TAkka Library Design	25
3.3	Type-parameterized Actor	25
3.4	Actor Reference	26
3.5	Backward Compatible Hot Swapping	29
3.6	Supervisor Strategies	30
3.7	Design Alternatives	33
Chapter 4	Evolution, Not Revolution	35
4.1	Akka actor in Akka system	35
4.2	TAkka actor in TAkka system	35
4.3	TAkka actor in Akka system	35
4.4	Akka Actor in TAkka system	36
Chapter 5	TAkka: Evaluation	39
5.1	Wadler's Type Pollution Problem	39
5.2	Expressiveness and Correctness	40
5.3	Efficiency, Throughput, and Scalability	41
5.4	Assessing System Reliability	42
Chapter 6	Future Work	54

Abstract

abstract abstract

Chapter 1

Introduction

blah blah blah

1.1 Contributions

The overall goal of the thesis is to develop a framework that makes it possible to construct reliable distributed applications written using and verified by our libraries which merges the advantages of type checking and the supervision principle. The key contributions of this thesis are:

- The design and implementation of the TAKka library, where actors are parameterized by the type of messages it expects. The library (Chapter 3) mixes statical and dynamical type checking so that type errors are detected at the earliest opportunity. The library separates message types and message handlers for the purpose of supervision from those for the purpose of general computation. The decision is made so that type-parameterized actors can form a supervision tree. Interestingly, this decision coincides with our recommendation in the model analysis for improving availability. The TAKka library is carefully designed so that Akka programs can gradually migrate to their TAKka equivalents (evolution) rather than requiring providing type parameters everywhere (revolution). In addition, the type pollution problem can be straightforwardly avoided in TAKka.
- A framework for evaluating libraries that support the supervision principle. The evaluation (Chapter 5) compares the TAKka library and the Akka library in terms of expressiveness, efficiency and scalability. Results show that TAKka applications add minimal runtime overhead to the underlying Akka system and have a similar code size and scalability compared to their Akka equivalents. Finally, we port the Chaos Monkey library

for testing the supervision relationship and design a Supervision View library for dynamically capturing the structure of supervision trees. We believe that similar techniques can be applied to Erlang and new libraries that support the supervision principle.

Chapter 2

Background and Related Work

2.1 Actor Programming

The Actor Programming Model is first formalised by Hewitt et al. [1973] for the purpose of constructing concurrent systems. In the model, a concurrent system consists of actors which are primitive computational components. Actors communicate with each other by sending messages but independently react to messages they receive.

The actor model defined in [Hewitt et al., 1973] does not specify its implementation strategies. Popular modern programming languages and libraries that support the actor model includes, Erlang [Ericsson AB., 2013a], Akka [Typesafe Inc. (b), 2012] and CloudHaskell [Epstein et al., 2011].

2.2 The Supervision Principle

The core idea of the supervision principle is that actors should be monitored and restarted when necessary by their supervisors in order to improve the availability of a software system. The supervision principle is first proposed in the Erlang OTP library [Ericsson AB., 2013c] and is adopted by the Akka library [Typesafe Inc. (b), 2012].

A supervision tree in Erlang consists of two types of actors: workers and supervisors. A worker implements part of the business logic and reacts to request messages. A supervisor is responsible for initializing and monitoring its children, which are workers or supervisors for other actors, and restarting its children when necessary. The behaviour of a supervisor is defined by its *supervision strategy*.

The Akka library makes supervision obligatory. In Akka, every user-created actor is either a child of the system guidance actor or a child of another user-

created actor. Therefore, every Akka actor is potentially the supervisor of some other actors. Different from the Erlang system, an Akka actor can be both a worker and a supervisor.

2.3 The Erlang Programming Language

Erlang [Ericsson AB., 2013a] is a dynamically typed functional programming language used for constructing concurrent programs. It is widely used in scalable real-time systems. The high availabilities of Erlang systems are largely attributed to the five OTP design principles, namely, supervision trees, behaviours, applications, releases, and release handling [Ericsson AB., 2013c]. This section gives a brief introduction to the Erlang programming language and OTP design principles, based on related material in Ericsson AB. [2013a], Armstrong [2007] and Ericsson AB. [2013c].

2.3.1 Actor Programming in Erlang

(This section summarises material from [Armstrong, 2007, Chapter 8] and [Ericsson AB., 2013b, Chapter 3])

An Erlang application consists of one or more module files, each of which defines a set of functions. The notion of the Erlang *process* minimizes the gap between sequential programming and concurrent programming. In Erlang, a process is a thread of function execution. It can receive messages of any type via its process identifier (pid). Defining an Actor in Erlang is as simple as providing a receive block to the body of the function spawned in a process.

2.3.1.1 Processes Creation

A *process* in Erlang is a thread of function execution. A process is created by calling the `spawn` method. Figure 2.1 gives the API of the `spawn` method [Ericsson AB., 2013a]. Calling `spawn(Module, Function, Args)` creates a process that executes the function `Module:Function(Args)`, where `Args` is a list of arguments. The `spawn` method returns a process identifier (pid) of the created process, which terminates at the end of executing the function.

To demonstrate the creation and usage of Erlang process, Figure 2.2 shows the echo example ¹ in Ericsson AB. [2013b] and its test result. As the result shows, the `say_something` function prints out its first argument for the number

¹The `tut14` module found in the Getting Started with Erlang Users Guide (version 5.10.3)

```

1 spawn(Module, Function, Args) -> pid()
2   Module = Function = atom()
3   Args = [Arg1,...,ArgN]
4   ArgI = term()

```

Figure 2.1: Erlang API: spawn

of times specified by its second argument. What is more interesting is the result of `echo:start()`, which spawns two processes. The result shows that the execution of the two processes and the main thread, which returns a pid (i.e. `<0.41.0>`), are in parallel.

```

1 -module(echo).
2
3 -export([start/0, say_something/2]).
4
5 say_something(_, 0) ->
6   done;
7 say_something(What, Times) ->
8   io:format("~p~n", [What]),
9   say_something(What, Times - 1).
10
11 start() ->
12   spawn(echo, say_something, [hello, 3]),
13   spawn(echo, say_something, [goodbye, 3]).
14
15 %% Terminal Output:
16 %% 1> c(echo).
17 %% {ok,echo}
18 %% 2> echo:say_something(hello, 3).
19 %% hello
20 %% hello
21 %% hello
22 %% done
23 %% 3> echo:start().
24 %% hello
25 %% goodbye
26 %% <0.41.0>
27 %% hello
28 %% goodbye
29 %% hello
30 %% goodbye

```

Figure 2.2: Erlang Example: An Echo Process

2.3.1.2 Message Passing

In the echo example, the two processes are executed independently. In Erlang, users can send a message to a process via its pid using the `!` primitive. For example, the code

```
1 Pid ! Msg
```

will send the message `Msg` to the process whose pid is `Pid`. Messages sent to a process is queued in the mailbox of the recipient. To handle a received message, a process provides a receive block with the following syntax:

```
1 receive
2   Message1 [when Guard1] ->
3     Action1 ;
4   Message2 [when Guard2] ->
5     Action2 ;
6   ...
7   MessageN [when GuardN] ->
8     ActionN
9 end
```

Figure 2.3: Erlang API: receive

In the Erlang API given in Figure 2.3, `receive` and `end` are primitives that denotes the scope of the receive block. The receive block defines a set of guarded message patterns to which the current processed message will be matched in order. If the current message matches a pattern, the corresponding action will be evaluated. If the current message does not match any pattern, it will be saved in the mailbox and the next message in the mailbox will be processed. When reaching a receive block, the evaluation of the process will be suspended until at least one message in the mailbox matches one of the guarded patterns.

The `echo_actor` example, given in Figure 2.4, spawns two processes, both of which verbatim print the received messages. The `print` message terminates as soon as the first message is processed. On the contrary, the receive block of the `loop` process is recursively defined so that the process can always processing incoming messages. Line 34 of Figure 2.4 reveals 2 properties of message passing in Erlang. Firstly, message sending is always a successful operation that returns the value of the sent message. At line 24, `P` is a pid pointing to a terminated process. Nevertheless, sending a message to `P` is still permitted. Secondly, message sending is an asynchronous operation. In this example, the evaluation result of line 23 is printed out after the evaluation result of line 24,

probably because it takes some time to match the message sent in line 23.

```
1 -module(echo_actor).
2
3 -export([start/0, loop/0, print/0]).
4
5 loop() ->
6     receive
7         Msg ->
8             io:format("loop: ~p~n", [Msg]),
9             loop()
10    end.
11
12 print() ->
13     receive
14         Msg ->
15             io:format("print: ~p~n", [Msg])
16    end.
17
18 start() ->
19     L = spawn(echo_actor, loop, []),
20     P = spawn(echo_actor, print, []),
21     L ! hello1,
22     P ! hello2,
23     L ! hello3,
24     P ! hello4.
25
26
27 %% Terminal Output:
28
29 %% 1> c(echo_actor).
30 %% {ok,echo_actor}
31 %% 2> echo_actor:start().
32 %% loop: hello1
33 %% print: hello2
34 %% hello4
35 %% loop: hello3
```

Figure 2.4: Erlang Example: An Echo Actor

2.3.1.3 An Erlang Actor with State

Erlang is a functional programming language where the value of a variable is immutable once assigned. On the other side, the result of a computation, for example, a search query, often depends on the value of some internal states. Therefore, an Erlang actor needs to retain or update its internal states when

it update its behaviour. One common method to define an Erlang actor with internal state is passing the state to the behaviour function.

The counter example defined in Figure 2.5 has one state variable, `Val`, which records the number of messages it has processed. The internal state is initialized to 0 when the actor is created at line 5. The value of the state is incremented each time when a message is processed (line 14 and line 17).

```
1 -module(counter).
2 -export([start/0, counter/1]).
3
4 start() ->
5     S = spawn(counter, counter, [0]),
6     S ! increment,
7     send_msgs(S, 3),
8     S.
9
10 counter(Val) ->
11     receive
12         increment ->
13             io:fwrite("increase counter to ~w~n", [Val+1]),
14             counter(Val+1);
15         Msg ->
16             io:fwrite("~w message(s) that has/have been processed ~n",
17                 [Val+1]),
18             counter(Val+1)
19     end.
20
21 send_msgs(_, 0) -> true;
22 send_msgs(S, Count) ->
23     S ! "Hello",
24     send_msgs(S, Count-1).
25
26 %% Terminal Output:
27 %% 1> c(counter).
28 %% {ok,counter}
29 %% 2> counter:start().
30 %% increase counter to 1
31 %% <0.95.0>
32 %% 2 message(s) has/have been processed
33 %% 3 message(s) has/have been processed
34 %% 4 message(s) has/have been processed
```

Figure 2.5: Erlang Example: A Message Counter

2.3.2 Supervision in Erlang

(This section summarises material from [Ericsson AB., 2013c, Chapter 5])

Supervision tree is probably the most important concept in the OTP design principle [Ericsson AB., 2013c]. A supervision tree consists of workers and supervisors. Workers are processes which carry out actual computations while supervisors are processes which inspect a group of workers or sub-supervisors. Since both workers and supervisors are processes and they are organised in a tree structure, the term *child* is used to refer to any supervised process in literature. An example of the supervision tree is presented in Figure 2.6 ¹, where supervisors are represented by squares and workers are represented by circles.

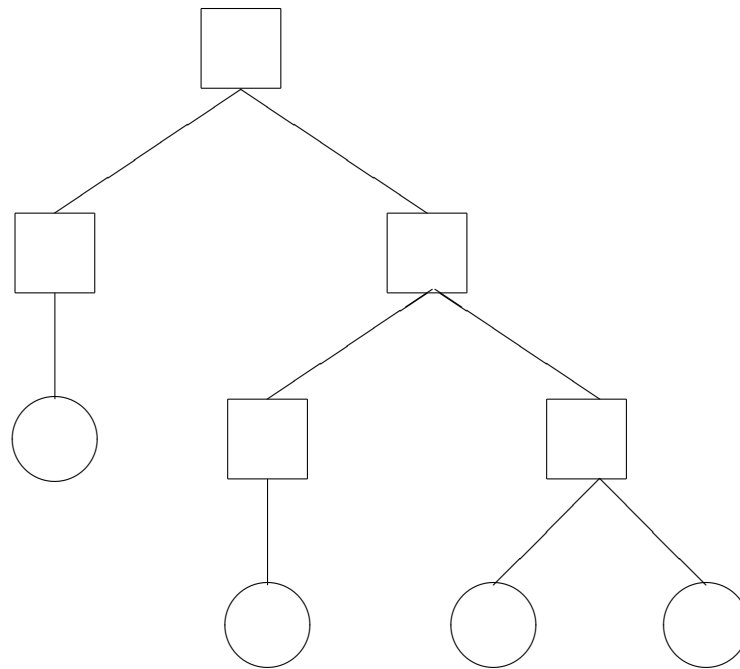


Figure 2.6: a supervision tree

2.3.2.1 Restart Strategy

2.3.2.2 Child Specification

In principle, a supervisor is accountable for starting, stopping and monitoring its child processes according to a list of *child specifications*. A child specification

¹This example is cited from OTP Design Principles/Overview. Restart strategies of each supervisor, however, are removed from the figure since they are not related to the central ideas discussed at this moment.

```

1 Module:init(Args) -> Result
2
3 Args = term()
4 Result = {ok, {{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
5     RestartStrategy = strategy()
6     MaxR = integer()>=0
7     MaxT = integer()>0
8     ChildSpec = child_spec()
9
10 child_spec() =
11     {Id :: child_id(),
12      StartFunc :: mfargs(),
13      Restart :: restart(),
14      Shutdown :: shutdown(),
15      Type :: worker(),
16      Modules :: modules()}
17
18 child_id() = term()
19
20 mfargs() =
21     {M :: module(), F :: atom(), A :: [term()] | undefined}
22
23 restart() = permanent | transient | temporary
24
25 shutdown() = brutal_kill | timeout()
26
27 strategy() = one_for_all
28             | one_for_one
29             | rest_for_one
30             | simple_one_for_one
31
32 worker() = worker | supervisor
33
34 modules() = [module()] | dynamic

```

Figure 2.7: Erlang API: supervision

contains 6 pieces of information Ericsson AB. [2013c]: i) an internal name of the supervisor to identify the child. ii) the function call to start the child process. iii) whether the child process should be restarted after the termination of its siblings or itself. iv) how to terminate the child process. v) whether the child process is a worker or a supervisor. vi) a singleton list which specifies the name of the callback module.

2.3.2.3 Starting a Supervisor

2.3.2.4 Adding and Stopping a Child Process

2.3.3 An Erlang Example

2.3.4 OTP Design Principles

Based on 10 years of experience of using Erlang in Enterprise level applications, Erlang developers summarized 5 OTP design principles in 1999 to improve the reliability of Erlang applications Ericsson AB. [2013c]: The Behaviour Principle, The Application Principle, The Release Principle, The Release Handling Principle, and The Supervision Principle. The Supervision Principle has been introduced in the previous section. This section describes the idea of the remaining 4 OTP design principles and the methodology of applying them in a JVM based environment, such as Java and Scala. The Supervision principle, which is the central topic of this thesis, has no direct correspondence in native JVM based systems.

2.3.4.1 The Behaviour Principle

Behaviours in Erlang, like interface or traits in the objected oriented programming, abstracts common structures and patterns of process implementations. With the help of behaviours, Erlang code can be divided into a generic part, a behaviour module, and a specific part, a callback module. Most processes, including the supervisor in Section 2.3.2, could be implemented by realising a set of pre-defined callback functions for one or more behaviours. Although ad-hoc code and programming structures may be more efficient, using consistent general interfaces make code more maintainable and reliable. Standard Erlang/OTP behaviours include:

- *gen_server* for constructing the server of a clientserver paradigm.
- *gen_fsm* for constructing finite state machines.

- *gen_event* for implementing event handling functionality.
- *supervisor* for implementing a supervisor in a supervision tree.

Last but not least, users could define their own behaviours in Erlang.

♠example ♠

2.3.4.2 The Application Principle

The OTP platform is made of a group of components called applications. To define an application, users need to annotate the implementation module with “-behaviour(application)” statement and implement the *start/2* and the *stop/1* functions. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process ¹.

Distributed applications may be deployed on several distributed Erlang nodes. An Erlang distributed application will be restarted at another node when its current node goes down. A distributed application is controlled by both the application controller and the distributed application controller ², both of which are part of the *kernel* application. Two configuration parameters must be set before loading and launching a distributed application. First, possible nodes where the distributed application may run must be explicitly pointed. Then, all nodes configured in the last step will be sent a copy of the same configuration which include three parameters: the time for other nodes to start, nodes that must be started in given time, and nodes that can be started in a given time.

2.3.4.3 The Release Principle

2.3.4.4 The Release Handling Principle

2.3.4.5 Applying OTP Design Principles Outside Erlang

¹registered as *application_controller*

²registered as *dist.ac*

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

Table 2.1: Using OTP Design Principles in JAVA/Scala Programming

2.4 The Akka Library

Akka is the first library that makes supervision obligatory. The API of the Akka library [Typesafe Inc. (a), 2012; Typesafe Inc. (b), 2012] is similar to the Scala Actor library [Haller and Odersky, 2006, 2007], which borrows syntax from the Erlang languages [Armstrong, 2007; Ericsson AB., 2013a]. Both Akka and Scala Actor are built in Scala, a typed language that merges features from Object-Oriented Programming and Functional Programming. This section gives a brief tutorial on Akka, based on related materials in the Akka Documentation Typesafe Inc. (b) [2012].

2.4.1 Actor Programming in Akka

(This section summarises material from [Typesafe Inc. (b), 2012, Section 2.3 and 3.1])

2.4.1.1 Actor System

Akka actors resident in Actor Systems. An actor system organises related actors in a tree structure and provides services such thread scheduling, network connection, and logging. In Akka, an actor is an independent computational component; related actors consist an actor system that solves a complex computational logic; one or several actor systems consist a complete application. To create an actor system, users provide a name and an optional configuration to the `ActorSystem` constructor as follows.

```

1 val defaultsystem = ActorSystem("defaultsystem")
2 val advancedsystem = ActorSystem("advancedsystem", configure)

```

2.4.1.2 The Actor Class

A simple Akka actor consists of its internal state, a behavior function and contextual information for that actor. In the following code, we define a simple actor that reports the number of messages it has received.

```
1 import akka.actor.Actor
2
3 class MessageCounter extends Actor {
4   int counter = 0
5   def receive = {
6     case m =>
7       counter = counter + 1
8       println(counter + " messages have been counted.")
9   }
10 }
```

Figure 2.8: Akka Example: A Message Counter

An Akka actor may contain some mutable variables represents its *internal state*. The `MessageCounter` has one state variable, `counter`, which is initialized to 0 and is incremented each time when a message is processed. When the actor is restarted by its supervisor, either due to a failure caused by itself, or be enforced by its supervisor for other reasons, the internal state is reset to the initial value.

♠discuss the downside of mutable state in the design alternative section

♠

The *behavior* of an actor is a function that reacts incoming messages. An Akka actor must define a `receive` method as its initial behaviour. To update its behaviour at run-time, an actor can pass a new behavior function to the `become` method, or call the `unbecome` method to roll back to its previous behavior, if there is one. The behavior of an actor is reset to the initial one when it is restarted.

The `context` field is created automatically to record *contextual information* for the respected actor. The `context` field is used to create child actors, record the actor system that the actor belongs to, and hot swap the behavior.

2.4.1.3 Props

A `Props` instance specifies the configuration of creating an actor. A `Props` is an immutable value and therefore, once created, can be consistently shared across

the potentially distributed system. An instance of Props can be created in one of the following two ways.

```
1 import akka.actor.Props
2
3 val props1 = Props[MyActor]
4 val props2 = Props(classOf[MyActor], "arg")
```

In the first example (line 3), a Props instance is created by calling the non-argument constructor of the MyActor class. In the second example (line 4), a Props instance is created by passing a string value to a constructor of MyActor that takes a string parameter. More arguments can be sent to the constructor of MyActor if it requires more parameters. The signature of the constructor, including the number, types and order of its parameters, is verified at the run time. If no matched constructor is found when initializing the Props object, an IllegalArgumentException will arise.

2.4.1.4 Mailbox

An actor receives messages from other parts of the application. Arrived messages are queued in its sole mailbox to be processed. Different from the Erlang design (Section 2.3.1.2), the behavior function of an Akka actor must be able to process the message it is given. If the message does not matches any message pattern of the current behavior, a failure arises.

Undefined messages are treated differently in different Akka versions. In versions prior to 2.0, an Akka actor raises an exception when it processes an undefined message. It means that sending an ill-typed message will cause a failure at the receiver side. In Akka 2.1, 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. In Akka 2.2, the Stash trait can be used to pending the process of a particular message until the current behaviour is updated.

2.4.1.5 Actor Reference and Actor Path

Actors collaborate by sending messages to each other. In Akka, an actor resides in the Actor System where it is created. On the other hand, an actor reference, an instance of ActorRef, is used by other parts of the application to send messages to the actor it represents. To send a message, msg, to an actor reference, actorRef, a user simply call the ! method of the actor reference. In

Akka Scala API, the above operation can be coded as:

```
1 actorRef ! msg
```

which is the syntactic sugar for

```
1 actorRef.!(msg)
```

An actor path is a symbolic representation of the address where an actor can be located. Since actors forms a tree hierarchy in Akka, a unique address can be allocated for each actor by appending a non-conflict actor name to the address of its parent. Examples of akka addresses are:

```
1 "akka://mysystem/user/service/worker"           //local
2 "akka.tcp://mysystem:example.com:1234/user/service/worker" //remote
3 "cluster://mycluster/service/worker"           //cluster
```

The first address represents the path to a local actor. Like a URI, the address consists of the scheme name (akka), actor system name (e.g. `mysystem`), and names of actors from the guardian actor (`user`) to the respected actor (e.g. `service`, `worker`). The second address represents the path to a remote actor. In addition to components of a local address, a remote address further specifies the communication protocol (`tcp` or `udp`), the IP address or domain name (e.g. `example.com`), and the port number (e.g. `1234`) used by the actor system to receive messages. The third address represents the desired format of a path to an actor in a cluster environment in a further Akka version. In the design, protocol, IP/domain name, and port number are omitted in the address of an actor which may transmit around the cluster or have multiple copies.

An actor path corresponds to an address where an actor can be identified. It can be initialized without the creation of an actor. Moreover, an actor path can be re-used by a new actor after the termination of an old actor. Two actor paths are considered equivalent as long as their symbolic representations are equivalent strings. On the contrary, an actor reference must correspond to an existing actor, either an alive actor located at the corresponding actor path, or the special `DeadLetter` actor which receives messages sent to terminated actors. Two actor references are equivalent if they correspond to the same actor path and the same actor. An restarted actor is considered as the same actor as the one before the restart because the life cycle of an actor is not visible to the users of `ActorRef`.

2.4.1.6 Creating Actors and Obtaining Actor References

An actor is created by passing a Props to the `actorOf` method provided by ActorSystem and ActorContext. Calling `system.actorOf(...)` creates an actor supervised by the system guidance actor for user-created actors (user). Calling `context.actorOf(...)` creates an actor supervised by the current actor.

Once an actor is created, there are 2 ways to obtain a reference to that actor. The first method passing the return value of `actorOf` to a variable so that the returned actor reference can be reused. The second method is inquiring on the actor address via the `actorFor` method in versions prior to 2.1 or the `actorSelection` method in version 2.2.

2.4.1.7 Hot Swapping on Actor Behaviour

As mentioned in Section 2.4.1.2, the behaviour of an actor can be updated at the run-time. Behaviours of Akka actor are kept in a stack. A behaviour function is pushed to the stack when `context.become(...)` is called; and is popped from the stack when `context.unbecome()` is called. To demonstrate hot swapping on the behaviour of Akka actors, Figure 2.9 gives an example modified from [Typesafe Inc. (b), 2012, Section 3.1]. The test output shows that the behaviour function has been updated each time when a Swap message is processed.

2.4.2 Supervision in Akka

(This section summarises material from [Typesafe Inc. (b), 2012, Section 2.4 and 3.4])

The Akka library makes supervision obligatory by restricting the way of creating actors. Every user-created actor is created as either a child of the system guardian actor or a child of another user-created actor. Therefore, all user-created actors in an actor system, together with the guardian actor of that actor system, form a tree structure. Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance. This section summarises concepts in the Akka supervision tree.

2.4.2.1 Children

Every actor in Akka is a supervisor for a list of other actors. An actor creates a new child by calling `context.actorOf(...)` and removes a child by calling

```

1 import akka.actor._
2 import akka.event.Logging
3
4 case object Swap
5
6 class Swapper extends Actor{
7   import context._
8   val log = Logging(system, this)
9
10  def receive = {
11    case Swap =>
12      log.info("Hi")
13      become({
14        case Swap =>
15          log.info("Ho")
16          unbecome()}, discardOld=false)
17  }
18 }
19
20 object SwapperApp extends App {
21   val system = ActorSystem("SwapperSystem")
22   val swap = system.actorOf(Props[Swapper], "swapper")
23   swap ! Swap
24   swap ! Swap
25   swap ! Swap
26   swap ! Swap
27 }
28
29 /* Terminal output:
30 Hi
31 Ho
32 Hi
33 Ho
34 */

```

Figure 2.9: Akka Behaviour Swap Example


```

context.stop(child).

1
2 val system = ActorSystem("mysystem")
3 val actor = system.actorOf(Props[MyActor], "myactor")

1 class MyActor extends Actor {
2   val child = context.actorOf(Props[Child], "mychild")
3   // recieve etc.
4 }

```

2.4.2.2 Supervisor Strategy

The Akka library implements two supervisor strategies: `OneForOne` and `AllForOne`. The `OneForOne` supervisor strategy corresponds to the `one_for_one` supervision strategy in OTP, which restart a child when it fails. The `AllForOne` supervisor strategy corresponds to the `one_for_all` supervision strategy in OTP, which restart all children when any of them fails. The `rest_for_all` supervision strategy in OTP has no corresponds in Akka because Akka actor does not specify the initialization order of its children. Simulating the `rest_for_all` strategy in Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. It is not clear whether the lack of the `rest_for_one` strategy will result in difficulties when rewriting Erlang applications in Akka.

```

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

```

Figure 2.10: Akka Supervisor Strategies

Figure 2.10 gives API of Akk supervisor strategies. As in OTP, for each supervisor strategy, users can specify the maximum number of restarts permitted for its children within a period. The default supervisor strategy in Akka is `OneForOne` that permits unlimited restarts.

As shown in the API, an Akka supervisor strategy can choose different reactions for different reasons of child failures in its decider parameter. Recall that `Throwable` is the superclass of `Error` and `Exception` in Scala and Java. Therefore, users can pattern match on possible types and values of `Throwable` in the decider function. In other words, when the failure of a child is passed

to the decider function of the supervisor, it is matched to a pattern that reacts to that failure.

The `decider` function contains user-specified computations and returns a value of `Directive` that denotes the standard recovery process implemented by the Akka library developers. The `Directive` trait is an enumerated type that has four possible values: the `Escalate` action which throws the exception to the supervisor of the supervisor, the `Restart` action which replaces the failed child with a new one, the `Resume` action which asks the child to process the message again, and the `Stop` action which terminates the failed actor permanently.

2.4.3 An Akka Example

Figure 2.11 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 raise. 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` raises. 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 is delivered. The last message is not processed because the both calculators a terminated because the simple calculator fails more frequently than allowed.

```

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:
43   Actor[akka://MySystem/user/safecalculator]
44 10 / 5 = 2
45 java.lang.ArithmeticException: / by zero
46 ArithmeticException Raised to:
47   Actor[akka://MySystem/user/safecalculator]
48 java.lang.ArithmeticException: / by zero
49 3 * 2 = 6
50 ArithmeticException Raised to:
51   Actor[akka://MySystem/user/safecalculator]
52 java.lang.ArithmeticException: / by zero
53 */

```

Figure 2.11: Supervised Calculator

Chapter 3

TAkka: Design and Implementation

3.1 Mixing Static and Dynamic Type Checking

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

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

3.1.1 Scala Type Descriptors

Scala 2.8 defines a `Manifest` class¹ 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.

¹Scala 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.

```

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,
16 Int]]

```

3.1.2 Typed Name Server

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

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

A typed name, `TSymbol`, is a name shipped with a type descriptor. A typed value, `TValue`, is a value shipped with a type descriptor, which describes a super type of the most precise type of that value. In Scala, `TSymbol` and `TValue` can be simply defined as in Figure 3.1:

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

of `TSymbol`. `TValue` is declared as a *case class* for the same reason.

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

```

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 3.1: TSymbol and TValue

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

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

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

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

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

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

Notice that unset and get operations succeed as long as the associated type of the input name is the supertype of the associated type of the registered name. To permit polymorphism, the hashCode method of TSymbol defined in Figure 3.1 does not take type values into account. Equivalence comparison on TSymbol instances, however, should consider the type. Although the notion of TValue does not appear in the API, it is required for an efficient library implementation because the type information in TSymbol is ignored in the hashmap. Overriding the hash function of TSymbol also prevents the case where users accidentally register two typed names with the same symbol but different types, in which case if one type is a supertype of the other, the return value of get can be non-deterministic. Last but not least, when an operation fails, the name server returns false or None rather than raising an exception so that it is always available.

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

3.2 T Akka Library Design

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

3.3 Type-parameterized Actor

A T Akka actor has type `TypedActor[M]`. It inherits the Akka Actor trait to minimize implementation effort. Users of the T Akka library, however, do not need to use any Akka Actor APIs. Instead, we encourage programmers to use the typed fields given in Figure 3.2. Unlike other actor libraries, every T Akka actor class takes a type parameter `M` which specifies the type of messages it expects to receive. The same type parameter is used as the input type of the receive function, the type parameter of actor context and the type parameter of the actor self reference.

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

Figure 3.2: T Akka 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 3.6. The implementation of the `typedReceive` method, on the other hand, is always provided by users.

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

3.4 Actor Reference

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

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

T_Akka, it is clear that `ActorRef` is contravariant because `ActorRef[Operation]` is a subtype of `ActorRef[Division]` though `Division` is a subtype of `Operation`. contravariance is crucial to avoid the type pollution problem described at Section 5.1.

For ease of use, T_Akka provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method has three advantages. Firstly, explicit type conversion using `publishAs` is always type safe because the type of the result is a supertype of the original actor reference. Secondly, the semantics of `publishAs` does not


```

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 3.3: Actor Reference

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

3.4.1 Props and Actor Context

The type `Props` denotes the properties of an actor. A `Props` of type `Props[M]` is used when creating an actor of type `TypedActor[M]`. Say `myActor` is of type `MyActor`, which is a subtype of `TypedActor[M]`, a `Prop` of type `Prop[M]` can be created by one of the APIs in Figure 3.5:

Contrary to an actor reference, which is the interface for receiving messages, an actor context describes the actor’s view of the outside world. Because each actor is an independent computational primitive, an actor context is private to the corresponding actor. By using APIs in Figure 3.6, an actor can (i) retrieve an actor reference corresponding to a given actor path using the `actorFor` method, (ii) create a child actor with a system-generated or user-specified name using one of the `actorOf` methods, (iii) set a timeout denoting the time within which a new message must be received using the `setReceiveTimeout` method, and (iv) update its behaviours using the `become` method. 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

```

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 3.4: 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 3.5: Actor Props

```

1 abstract class ActorContext[M:Manifest] {
2   def actorOf [Msg] (props: Props[Msg])(implicit mt: Manifest[Msg]):
3   ActorRef[Msg]
4   def actorOf [Msg] (props: Props[Msg], name: String)(implicit mt:
5   Manifest[Msg]): ActorRef[Msg]
6   def actorFor [Msg] (actorPath: String)
7     (implicit mt: Manifest[Msg]): ActorRef[Msg]
8   def setReceiveTimeout(timeout: Duration): Unit
9
10  def become[SupM >: M](
11    newTypedReceive: SupM => Unit,
12    newSystemMessageHandler:
13      SystemMessage => Unit,
14    newSupervisorStrategy: SupervisorStrategy
15  )(implicit smt: Manifest[SupM]): ActorRef[SupM]
16 }

```

Figure 3.6: Actor Context

supervised by the current actor. Each actor created is assigned to a typed actor path, an Akka actor path together with a Manifest of the message type. Each actor system contains a typed name server. When an actor is created inside an actor system, a mapping from its typed actor path to its typed actor reference is registered to the typed name server. The actorFor method of ActorContext and ActorSystem fetches typed actor reference from the typed name server.

3.5 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 T Akka library runs. To support hot swapping on an actor's receive function, system message handler, and supervisor strategy, those three behaviour methods are maintained as object references.

The become method enables hot swapping on the behaviour of an actor. The become method in T Akka is different from behaviour upgrades in Akka in two aspects. Firstly, the supervisor strategy can be updated. In Akka, the supervisor strategy is an immutable value of an actor. We believe the supervisor strategy is an important behaviour of an actor and it should be as swappable as message handlers. Secondly, hot swapping in T Akka must be backward compatible. In other words, an actor must evolve to a version that is able to handle the original

message patterns. The above decision is made so that a service published to users will not be unavailable later.

```

1 abstract class ActorContext[M:Manifest] {
2   implicit private var mt:Manifest[M] = manifest[M]
3
4   def become[SupM >: M](
5     newTypedReceive: SupM => Unit,
6     newSystemMessageHandler:
7       SystemMessage => Unit
8     newSupervisorStrategy:SupervisorStrategy
9   )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] = {
10    val smt = manifest[SupM]
11    if (!(mt <:: smt))
12      throw BehaviorUpdateException(smt, mt)
13
14    this.mt = smt
15    this.systemMessageHandler = newSystemMessageHandler
16    this.supervisorStrategy = newSupervisorStrategy
17  }
18 }

```

Figure 3.7: Hot Swapping in T Akka

The become method is implemented as in Figure 3.7. 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.

3.6 Supervisor Strategies

The Akka library implements two of the three supervisor strategies in OTP: `OneForOne` and `AllForOne`. If a supervisor adopts the `OneForOne` strategy, a child will be restarted when it fails. If a supervisor adopts the `AllForOne` supervisor strategy, all children will be restarted when any of them fails. The third OTP supervisor strategy, `RestForOne`, restarts children in a user-specified order, and hence is not supported by Akka as it does not specify an order of initialization for children. Simulating the `RestForOne` supervisor strategy in

Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. None of the Erlang examples in Section 5 uses the `RestForOne` strategy. It is not clear whether the lack of the `RestForOne` strategy will result in difficulties when rewriting Erlang applications in Akka and TAKka.

Figure 3.8 gives APIs of supervisor strategies in Akka. As in OTP, for each supervisor strategy, users can specify the maximum number of restarts of any child within a period. The default supervisor strategy in Akka is `OneForOne` that permits unlimited restarts. `Directive` is an enumerated type with the following values: the `Escalate` action which throws the exception to the supervisor of the supervisor, the `Restart` action which replaces the failed child with a new one, the `Resume` action which asks the child to process the message again, and the `Stop` action which terminates the failed actor permanently.

None of the supervisor strategies in Figure 3.8 requires a type-parameterized classes during construction. Therefore, from the perspective of API design, both supervisor strategies are constructed in TAKka in the same way as in Akka.

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

Figure 3.8: Supervisor Strategies

3.6.1 Handling System Messages

Actors communicate with each other by sending messages. To maintain a supervision tree, a special category of messages should be handled by all actors. We define a trait ¹ `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`

¹A trait in Scala is similar to a JAVA abstract class, but trait permits multiple inheritance.

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

- `PoisonPill`

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

- `Restart`

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

- `ReceiveTimeout`

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

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

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

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

3.7 Design Alternatives

♠expend, with code ♠

Akka Typed Actor In the Akka library, there is a special class called `TypedActor`, which contains an internal actor and can be supervised. A service of `TypedActor` is invoked by method invocation instead of message exchanging. Code in Figure 3.9 demonstrates how to define a simple string processor using Akka typed actor. The Akka `TypedActor` prevent some type errors but have two limitations. Firstly, `TypedActor` does not permit hot swapping on its behaviours. Secondly, avoiding type pollution by using Akka typed actors is as awkward as using a plain object-oriented model, where supertypes need to be introduced. In Scala and Java, introducing a supertype in a type hierarchy requires modification to all affected classes.

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

Figure 3.9: Akka TypedActor Example

Actors with or without Mutable States The actor model formalized by Hewitt et al. [1973] does not specify its implementation strategy. In

Erlang, a functional programming language, actors do not have mutable states. In Scala, an object-oriented programming language, actors may have mutable states. The T Akka library is built on top of Akka and implemented in Scala. As a result, T Akka 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 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 use mutable state, 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]. In contrast, stateless services, e.g. RESTful web services, are more likely to achieve a good scalability and availability.

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

Chapter 4

Evolution, Not Revolution

Akka systems can be smoothly migrated to T Akka 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 ??) and how to use T Akka actors in a T Akka system (Figure 3.4). In the following, we will explain how to use T Akka actors in an Akka system and how to use an Akka actor in a T Akka system.

4.1 Akka actor in Akka system

4.2 T Akka actor in T Akka system

4.3 T Akka actor in Akka system

It is often the case that an actor-based library is implemented by one organization but used in a client application implemented by another organization. If a developer decided to upgrade the library implementation using T Akka 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 5.2, how would the upgrade affects client code, especially legacy applications built using the Akka library? T Akka 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 4.1, the actor implementation is upgraded to the TAkka version as in Figure 3.4. The client code, line 15 through line 25, is the same as the old Akka version as given in Figure ???. That is, no changes are required for the client application.

TAkka actor reference delegates the task of message sending to an Akka actor reference, its `untypedRef` field. In line 31 in Figure ??, 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 type may be sent to TAkka actors if Akka actor reference is used. As a result, users who are interested in the `UnhandledMessage` event may subscribe the event stream as in line 33.

4.4 Akka Actor in TAkka system

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

Users can initialize a TAkka actor reference by providing an Akka actor reference and a type parameter. In Figure 4.2, we re-use the Akka actor, initialize the actor in an Akka actor system, and obtained an Akka actor reference as in Figure ???. Then, we initialize a TAkka actor reference, `takkaServer`, which only accepts `String` messages.

```

1 class TakkaServerActor extends takka.actor.TypedActor[String] {
2   def typedReceive = {
3     case m:String => println("received message: "+m)
4   }
5 }
6
7 class MessageHandler(system: akka.actor.ActorSystem) extends
  akka.actor.Actor {
8   def receive = {
9     case akka.actor.UnhandledMessage(message, sender, recipient) =>
10      println("unhandled message:"+message);
11   }
12 }
13
14 object TakkaInAkka extends App {
15   val akkasystem = akka.actor.ActorSystem("AkkaSystem")
16   val akkaserver = akkasystem.actorOf(
17     akka.actor.Props[TakkaServerActor], "server")
18
19   val handler = akkasystem.actorOf(
20     akka.actor.Props(new MessageHandler(akkasystem)))
21
22   akkasystem.eventStream.subscribe(handler,
23     classOf[akka.actor.UnhandledMessage]);
24   akkaserver ! "Hello Akka"
25   akkaserver ! 3
26
27   val takkasystem = takka.actor.ActorSystem("TAkkaSystem")
28   val typedserver = takkasystem.actorOf(
29     takka.actor.Props[String, ServerActor], "server")
30
31   val untypedserver = takkaserver.untypedRef
32
33   takkasystem.system.eventStream.subscribe(
34     handler, classOf[akka.actor.UnhandledMessage]);
35
36   untypedserver ! "Hello TAkka"
37   untypedserver ! 4
38 }
39
40 /*
41 Terminal output:
42 received message: Hello Akka
43 unhandled message:3
44 received message: Hello TAkka
45 unhandled message:4
46 */

```

Figure 4.1: TAkka actor in Akka application

```

1 class AkkaServerActor extends akka.actor.Actor {
2   def receive = {
3     case m:String => println("received message: "+m)
4   }
5 }
6
7 object AkkaInTAkka extends App {
8   val system = akka.actor.ActorSystem("AkkaSystem")
9   val akkaserver = system.actorOf(
10     akka.actor.Props[AkkaServerActor], "server")
11
12   val takkaServer = new takka.actor.ActorRef[String]{
13     val untypedRef = akkaserver
14   }
15
16   takkaServer ! "Hello World"
17   // takkaServer ! 3
18 }
19
20 /*
21 Terminal output:
22 received message: Hello World
23 */

```

Figure 4.2: Akka actor in TAKka application

Chapter 5

TAkka: Evaluation

♠expend 19 examples into subsubsections. 4-5 pages expected. ♠

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

5.1 Wadler’s Type Pollution Problem

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

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

TArkka solves the type pollution problem by using polymorphism. Take the code template in Figure 5.1 for example. Let `V2CMessage` and `M2CMessage` be the type of messages expected from the View and the Model respectively. Both `V2CMessage` and `M2CMessage` are subtypes of `ControllerMsg`, which is the least general type of messages expected by the controller. In the template code, the controller publishes itself as different types to the view actor and the model actor. Therefore, both the view and the model only know the communication interface between the controller and itself. The `ControllerMsg` is a sealed trait

so that users cannot define a subtype of `ControllerMsg` outside the file and send the controller a message of unexpected type. Although type convention in line 25 and line 27 can be omitted, we explicitly use the `publishAs` to express our intention and let the compiler check the type. The code template is used to implement the Tik-Tak-Tok example in the T Akka code repository.

```

1 sealed trait ControllerMsg
2 class V2CMessage extends ControllerMsg
3 class M2CMessage extends ControllerMsg
4
5 trait C2VMessage
6 case class ViewSetController(controller:ActorRef[V2CMessage]) extends
7 C2VMessage
8 trait C2MMessage
9 case class ModelSetController(controller:ActorRef[M2CMessage]) extends
10 C2MMessage
11
12 class View extends TypedActor[C2VMessage] {
13   private var controller:ActorRef[V2CMessage]
14   // rest of implementation
15 }
16 Model extends TypedActor[C2MMessage] {
17   private var controller:ActorRef[M2CMessage]
18   // rest of implementation
19 }
20
21 class Controller(model:ActorRef[C2MMessage],
22                 view:ActorRef[C2VMessage]) extends
23 TypedActor[ControllerMessage] {
24   override def preStart() = {
25     model ! ModelSetController(
26       typedSelf.publishAs[M2CMessage])
27     view ! ViewSetController(
28       typedSelf.publishAs[V2CMessage])
29   }
30   // rest of implementation
31 }

```

Figure 5.1: Template for Model-View-Controller

5.2 Expressiveness and Correctness

Table 5.1 lists the examples used for expressiveness and correctness. We selected examples from Erlang Quiviq Arts et al. [2006] and open source Akka

projects to ensure that the main requirements for actor programming are not unintentionally neglected. Examples from Erlang Quiviq are re-implemented using both Akka and T Akka. Examples from Akka projects are re-implemented using T Akka. Following the suggestion in Hennessy and Patterson [2006], we assess the overall code modification and code size by calculating the geometric mean of all examples. The evaluation results in Table 5.2 show that when porting an Akka program to T Akka, about 7.4% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because T Akka code does not need to handle unexpected messages. On average, the total program size of Akka and T Akka 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 T Akka 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 those examples always passes instances of `HttpRequestEvent` to the `unapply` method and send the returned values to an actor that accepts messages of `HttpRequestEvent` type. Fortunately, the design flaw is exposed when upgrading the SOCKO implementation using T Akka.

5.3 Efficiency, Throughput, and Scalability

The T Akka 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 T Akka 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 ?? and Figure 3.4. When n ranges from 10^4 to 10^5 , the T Akka implementation is about 2 times slower as the Akka implementation. The cost of the last factor is close to the cost of transmitting

the string representation of fully qualified type names.

The JSON serialization example TechEmpower, Inc. [2013] is used to compare the throughput of 4 web services built using Akka Play, T Akka Play, Akka Socko, and T Akka Scoko. For each HTTP request, the example gives an HTTP response with pre-defined content. All web services are deployed to Amazon EC2 Micro instances (t1.micro), which has 0.615GB Memory. The throughput is tested with up to 16 EC2 Micro instances. For each number of EC2 instances, 10 rounds of throughput measurement is executed to gather the average and standard derivation of the throughput. The result reported in Figure 5.2 shows that web servers built using Akka-based library and T Akka-based library have similar throughput.

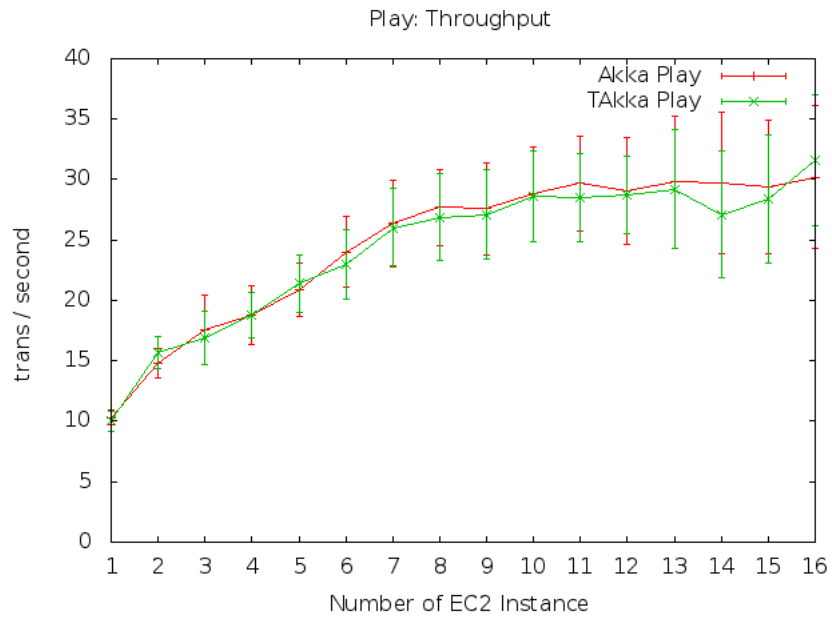
We further investigated the speed-up of multi-nodes T Akka applications by porting 6 micro benchmark examples, listed in Table 5.3, from the BenchErl benchmarks in the RELEASE project Boudeville et al. [2012]. Each BenchErl benchmark spawns one master process and many child processes for a tested task. Each child process is asked to perform a certain amount of calculation and report the result to the master process. The benchmarks are run on a 32 node Beowulf cluster at the Heriot-Watt University. Each Beowulf node comprises eight Intel 5506 cores running at 2.13GHz. All machines run under Linux CentOS 5.5. The Beowulf nodes are connected with a Baystack 5510-48T switch with 48 10/100/1000 ports.

Figure 5.3 and 5.4 reports the results of the BenchErl benchmarks. We report the average and the standard deviation of the run-time of each example. Depending on the ratio of the calculation time and the I/O time, benchmark examples scale at different levels. In all examples, T Akka and Akka implementations have almost identical run-time and scalability.

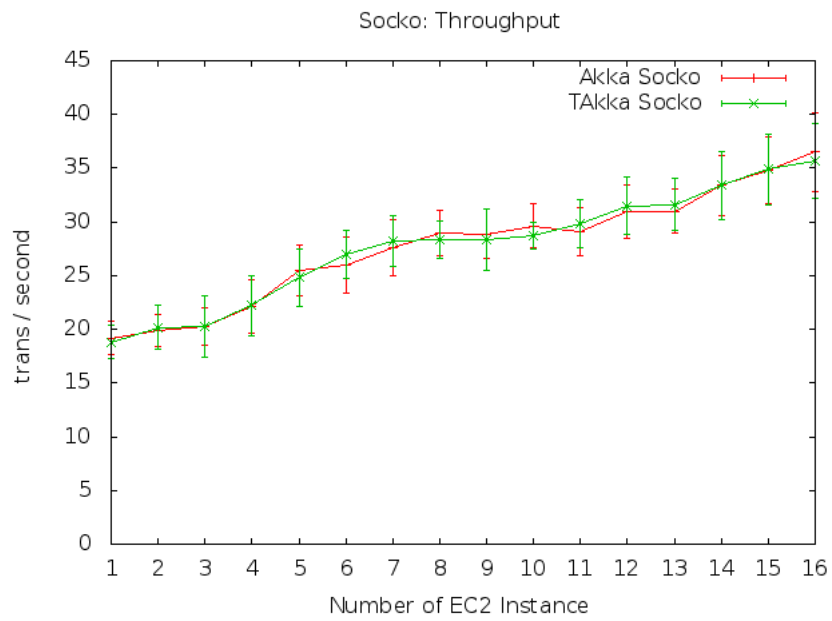
5.4 Assessing System Reliability

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?

T Akka is shipped with a Chaos Monkey library and a Supervision View library for assessing the reliability of T Akka applications. A Chaos Monkey test randomly kills actors in a supervision tree and a Supervision View test dy-

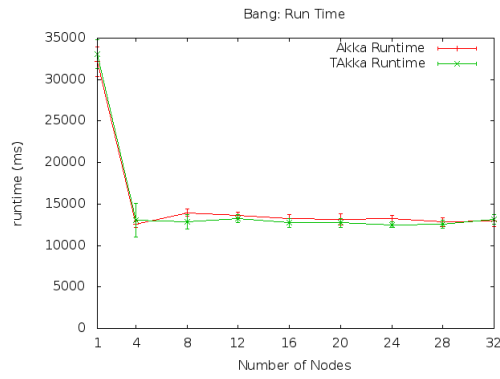


(a)

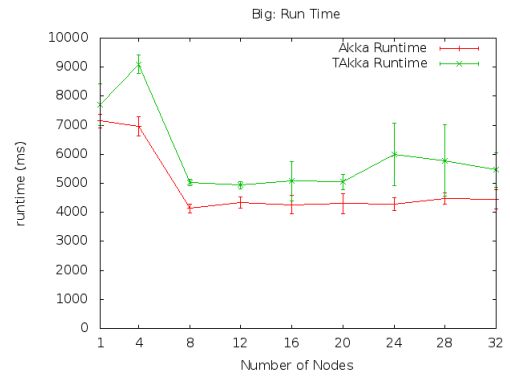


(b)

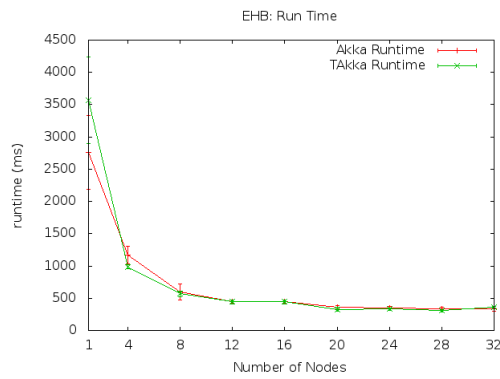
Figure 5.2: Throughput Benchmarks



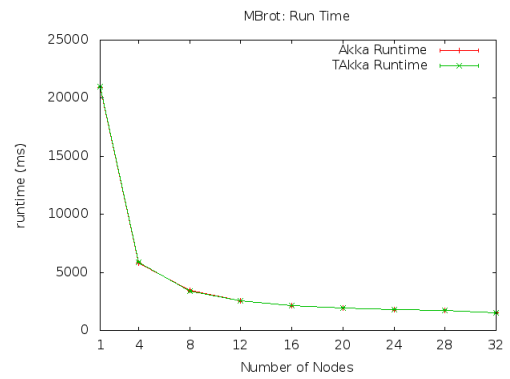
(a)



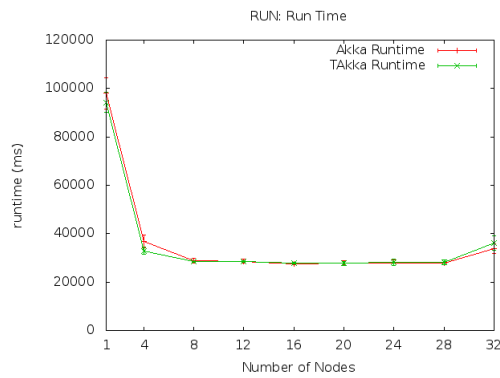
(b)



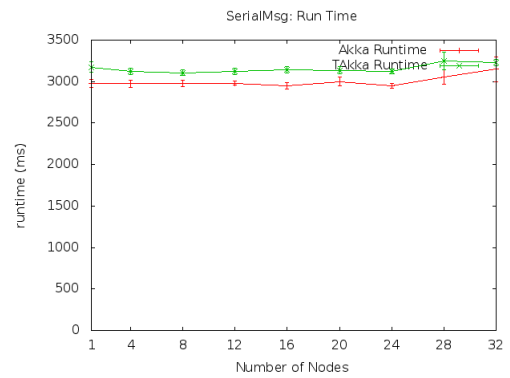
(c)



(d)

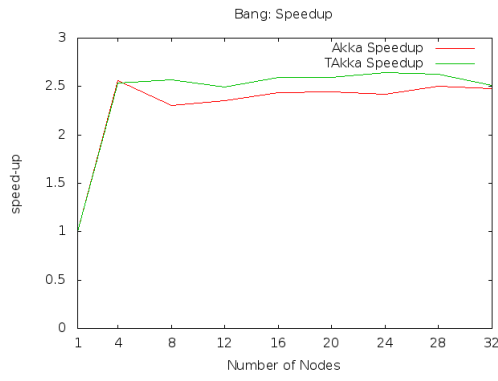


(e)

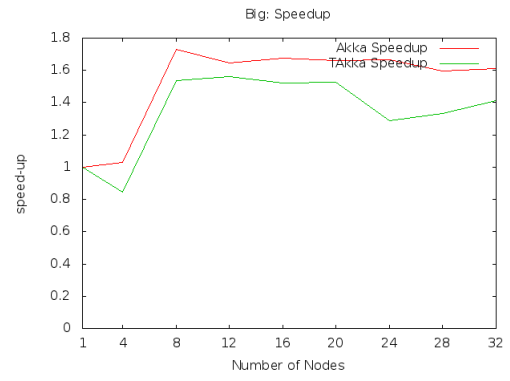


(f)

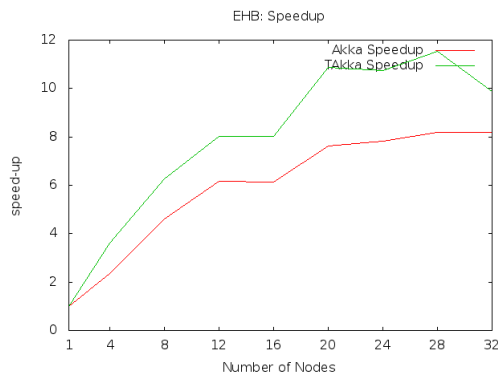
Figure 5.3: Runtime Benchmarks



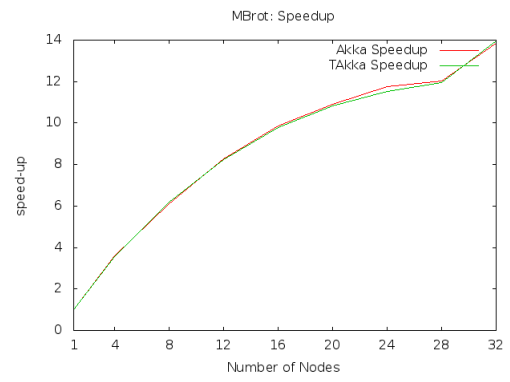
(a)



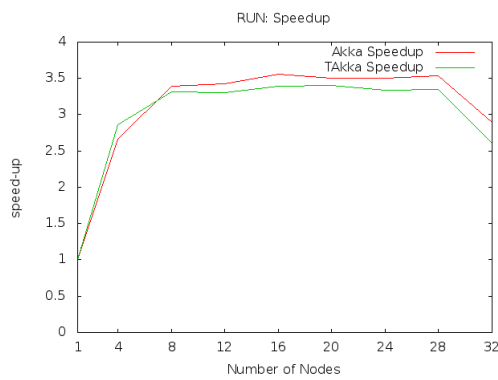
(b)



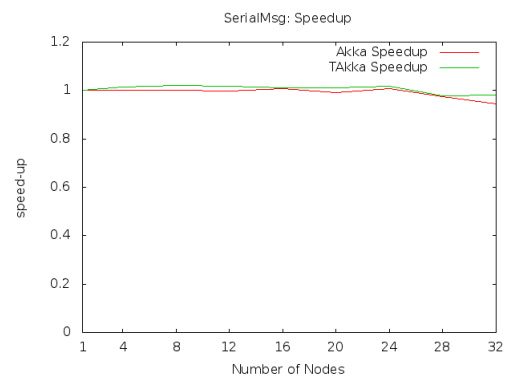
(c)



(d)



(e)



(f)

Figure 5.4: Scalability Benchmarks

namically 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 5.4.3) show that failures occur during the test. On the other hand, any failed actors are restored, and hence appropriately supervised applications (Section 5.4.4) pass Chaos Monkey tests.

5.4.1 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 5.4.

Figure 5.5 gives the API and the core implementation of TAKka Chaos Monkey. A user sets up a Chaos Monkey test by initializing a `ChaosMonkey` instance, defining the test mode, and scheduling the interval between each run. In each run, the `ChaosMonkey` instance sends a randomly picked actor a special message. When receive a Chaos Monkey message, a TAKka actor execute a corresponding potentially problematic code as described in Table 5.4. `PoisonPill` and `Kill` are handled by `systemMessageHandler` and can be overridden as described in Section 3.6.1. `ChaosException` and `ChaosNonTerminate`, on the other hand, are handled by the TAKka library and cannot be overridden.

5.4.2 Supervision View

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

A view master is initialized by calling one of the `apply` method of the

`ViewMaster` object as given in Figure 5.6. Each view master has an actor system and a master actor as its fields. The actor system is set up according to the given name and config, or the default configuration. The master actor, created in the actor system, has type `TypedActor[SupervisionViewMessage]`. After the start method of a view master is called, the view master periodically sends `SupervisionViewRequest` to interested nodes in supervision trees, where `date` is the system time just before the view master sends requests. When a `TAkka` actor receives `SupervisionViewRequest` message, it sends a `SupervisionViewResponse` message back to the view master and pass the `SupervisionViewRequest` message to its children. The `date` value in a `SupervisionViewResponse` message is the same as the `date` value in the corresponding `SupervisionViewRequest` message. Finally, the master actor of the view master records all replies in a hash map from `Date` to `TreeSet[NodeRecord]`, and sends the record to appropriate drawer on request.

5.4.3 A Partly Failed Safe Calculator

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

A test result, given in Figure 5.7, gives the expected tree structure at the beginning, 15 seconds and 30 seconds of the test. Figure 5.7(a) shows that the application initialized three safe calculators as described. In Figure 5.7(b), `S2` and its child are marked as dashed circles because it takes the view master more than 5 seconds to receive their responses. From the test result itself, we cannot tell whether the delay is due to a blocked calculation or a network congestion. Comparing to Figure 5.7(a), the child of `S3` is not shown in Figure 5.7(b) and Figure 5.7(c) because no response is received from it until the end of the test.

When the test ends, no response to the last request is received from S2 and its child. Therefore, both S2 and its child are not shown in Figure 5.7(c). S1 and its child appear in all three Figures because either they never fail during the test or they are recovered from failures within a short time.

5.4.4 BenchErl Examples with Different Supervisor Strategies

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

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

Table 5.1: Examples for Correctness and Expressiveness Evaluation

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

Table 5.2: Results of Correctness and Expressiveness Evaluation

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 5.3: Examples for Efficiency and Scalability Evaluation

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 5.4: TAcKa Chaos Monkey Modes

```

1 class ChaosMonkey(victims:List[ActorRef[_]],
    exceptions:List[Exception]){
2   private var status:Status = OFF;
3
4   def setMode(mode:ChaosMode);
5   def enableDebug();
6   def disableDebug();
7   def start(interval:FiniteDuration) = status match {
8     case ON =>
9   throw new Exception("ChaosMonkey is running: turn it off before
    restart it.")
10    case OFF =>
11      status = ON
12      scala.concurrent.future {
13        repeat(interval)
14      }
15  }
16  def turnOff()= {status = OFF}
17
18  private def once() {
19    var tempMode = mode
20    if (tempMode == Random){
21      tempMode = Random.shuffle(
22        ChaosMode.values.-(Random).toList).head
23    }
24    val victim = scala.util.Random.shuffle(victims).head
25    tempMode match {
26      case PoisonKill =>
27        victim.untypedRef ! akka.actor.PoisonPill
28      case Kill =>
29        victim.untypedRef ! akka.actor.Kill
30      case Exception =>
31        val e = scala.util.Random.shuffle(exceptions).head
32        victim.untypedRef ! ChaosException(e)
33      case NonTerminate =>
34        victim.untypedRef ! ChaosNonTerminate
35    }
36  }
37
38  private def repeat(period:FiniteDuration):Unit = status match {
39    case ON =>
40      once
41      Thread.sleep(period.toMillis)
42      repeat(period)
43    case OFF =>
44  }
45 }
46
47 object ChaosMode extends Enumeration {
48   type ChaosMode = Value
49   val Random, PoisonKill, Kill, Exception, NonTerminate = Value
50 }

```

Figure 5.5: Takka Chaos Monkey

```

1 sealed trait SupervisionViewMessage
2 case class SupervisionViewResponse(date:Date, reporterPath:ActorPath,
3 childrenPath:List[ActorPath]) extends SupervisionViewMessage
4 case class ReportViewTo(drawer:ActorRef[Map[Date,
5   TreeSet[NodeRecord]]]) extends
6   SupervisionViewMessage
7
8 case class SupervisionViewRequest(date:Date,
9 master:ActorRef[SupervisionViewResponse])
10 case class NodeRecord(receiveTime:Date, node:ActorPath,
11 childrenPath:List[ActorPath])
12
13 object ViewMaster{
14   def apply(name:String, config: Config, topnodes:List[ActorRef[_]],
15   interval:FiniteDuration):ViewMaster
16
17   def apply(name:String, topnodes:List[ActorRef[_]],
18   interval:FiniteDuration):ViewMaster
19
20   def apply(topnodes:List[ActorRef[_]],
21   interval:FiniteDuration):ViewMaster
22 }

```

Figure 5.6: Supersition View

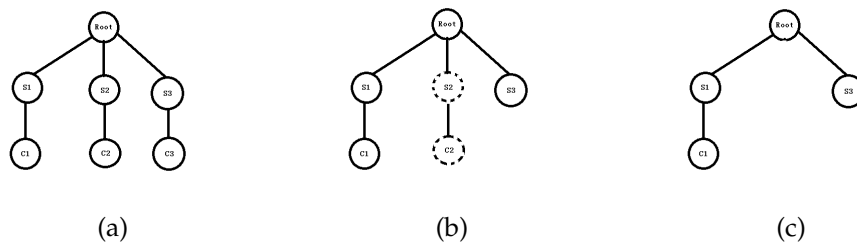


Figure 5.7: Supervision View Example

Chapter 6

Future Work

Chapter 7

Conclusion

Bibliography

- Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.
- Armstrong, Joe (2002). Concurrency Oriented Programming in Erlang. <http://112.ai.mit.edu/talks/armstrong.pdf>.
- Arts, T., Hughes, J., Johansson, J., and Wiger, U. (2006). 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. ACM.
- Boudeville, O., Cesarini, F., Chechina, N., Lundin, K., Papaspyrou, N., Sagonas, K., Thompson, S., Trinder, P., and Wiger, U. (2012). Release: a high-level paradigm for reliable large-scale server software. *Symposium on Trends in Functional Programming*.
- Doyle, C. and Allen, M. (2012). EnMAS: A new tool for multi-agent systems research and education. *Midwest Instruction and Computing Symposium*.
- Epstein, J., Black, A. P., and Peyton-Jones, S. (2011). Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell, Haskell '11*, pages 118–129, New York, NY, USA. ACM.
- Ericsson AB. (2013a). Erlang Reference Manual User’s Guide (version 5.10.3). <http://www.erlang.org>. Accessed on Oct 2013.
- Ericsson AB. (2013b). Getting Started with Erlang User’s Guide (version 5.10.3). http://www.erlang.org/doc/getting_started/users_guide.html. Accessed on Oct 2013.
- Ericsson AB. (2013c). OTP Design Principles User’s Guide (version 5.10.3). <http://www.erlang.org/doc/pdf/otp-system-documentation.pdf>.
- Excilys Group (2012). Gatling: stress tool. <http://gatling-tool.org/>. Accessed on Oct 2012.

- Fournet, C. and Gonthier, G. (2000). 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.
- Gilbert, S. and Lynch, N. (2002). Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59.
- Haller, P. and Odersky, M. (2006). Event-Based Programming without Inversion of Control. In Lightfoot, D. E. and Szyperski, C. A., editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 4–22.
- Haller, P. and Odersky, M. (2007). Actors that Unify Threads and Events. In Vitek, J. and Murphy, A. L., editors, *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, Lecture Notes in Computer Science (LNCS), pages 171–190. Springer.
- Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition.
- Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. pages 235–245.
- Imtarnasan, V. and Bolton, D. (2012). SOCKO Web Server. <http://sockoweb.org/>. Accessed on Oct 2012.
- Kuhn, R., He, J., Wadler, P., Bonér, J., and Trinder, P. (2012). Typed akka actors. private communication.
- Luna, D. (2013). Erlang Chaos Monkey. https://github.com/dLuna/chaos_monkey. Accessed on Mar 2013.
- Naftalin, M. and Wadler, P. (2006). *Java Generics and Collections*, chapter Chapter 5: Evolution, Not revolution. O’Reilly Media, Inc.
- Netflix, Inc. (2013). Chaos Home. <https://github.com/Netflix/SimianArmy/wiki/Chaos-Home>. Accessed on Mar 2013.
- Sangiorgi, D. and Walker, D. (2001). *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA.
- TechEmpower, Inc. (2013). Techempower web framework benchmarks. <http://www.techempower.com/benchmarks/>. Accessed on July 2013.

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

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

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

Zachrisson, M. (2012). Barbershop. <https://github.com/cyberzac/BarberShop>.
Accessed on Oct 2012.