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.

(*Jiansen HE*)

Table of Contents

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 The Actor Programming Model

The Actor Programming Model is first proposed 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. Each actor independently reacts to messages it receives.

The Actor model given in [Hewitt et al., 1973] does not specify its formal semantics and hence does not suggest implementation strategies neither. An operational semantics of the Actor model is developed by Gerif [Grief, 1975]. Baker and Hewitt [1977] later defines a set of axiomatic laws for Actor systems. Other semantics of the Actor model includes the denotational semantics given by Clinger [1981] and the transition-based semantic model by Agha [1985]. Meanwhile, the Actor model has been implemented in Act 1 [Lieberman, 1981], a prototype programming language. The model influences designs of Concurrency Oriented Programming Languages (COPLs), especially the Erlang programming language [Armstrong, 2007b], which has been used in enterprise-level applications since it was developed in 1986.

A recent trend is adding actor libraries to full-fledged popular programming languages that do not have actors built-in. Some of the recent actor libraries are: JActor [JActor Consulting Ltd, 2013] for the JAVA language, Scala Actor [Haller and Odersky, 2006, 2007] for Scala, Akka [Typesafe Inc. (b), 2012] for Java and Scala, and CloudHaskell [Epstein et al., 2011] for Haskell.

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 [Armstrong, 2007a,b] is a dynamically typed functional programming language originally designed at the Ericsson Computer Science Laboratory for implementing telephony applications [Armstrong, 2007a]. After using the Erlang language for in-house applications for ten years, when Erlang was released as open source in 1998, Erlang developers summarised five design principles shipped with the Erlang/OTP library, which stands for Erlang Open Telecom Platform [Armstrong, 2007a; Ericsson AB., 2013c].

In enterprise-level applications, Erlang typically collaborates with other languages to provide fault-tolerant support for distributed real-time applications. One of the early OTP applications, Ericsson's AXD 301 switch, is reported to have achieved nine "9"s availability, that is 99.9999999% of uptime, during the nine months experiment [Armstrong, 2002]. Up to the present, Erlang has been widely used in database systems (e.g. Mnesia, Riak, and Amazon SimpleDB) and messaging services (e.g. RabbitMQ and WhatsApp).

This section gives a brief introduction to the Erlang programming language and OTP design principles, based on related material in [Armstrong, 2007b] and [Ericsson AB., 2013a,b,c].

2.3.1 Actor Programming in Erlang

(This section summarises material from [Armstrong, 2007b, 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 for 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 when the execution of the function completes.

```
spawn(Module, Function, Args) -> pid()

  Module = Function = atom()
  Args = [Arg1,...,ArgN]
  ArgI = term()
```

Figure 2.1: Erlang API: spawn

To demonstrate the creation and usage of Erlang processes, Figure 2.2 shows the `echo` example modified from [Ericsson AB., 2013b, the `tut14` module] and its test result. As the terminal output shows, the `say_something` function prints out its first argument for the number 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 of the last `spawn` (i.e. `<0.41.0>`), are in parallel. As a consequence, the program prints out "hello", "goodbye", and the pid in a non-deterministic order.

```erlang
-module(echo).

-export([start/0, say_something/2]).

say_something(_, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(echo, say_something, [hello, 3]),
    spawn(echo, say_something, [goodbye, 3]).

%% Terminal Output:
%% 1> c(echo).
%% {ok,echo}
%% 2> echo:say_something(hello, 3).
%% hello
%% hello
%% hello
%% done
%% 3> echo:start().
%% hello
%% goodbye
%% <0.41.0>
%% hello
%% goodbye
%% hello
%% goodbye
```

Figure 2.2: Erlang Example: An Echo Process

2.3.1.2 Message Passing Style Concurrency

In the echo example, the two processes are executed independently. To be an actor, an Erlang process shall be able to receive messages from others and reacts to messages.

In Erlang, users can send a message to a process via its pid using the `!` primitive. For example, the code

```erlang
Pid ! Msg
```

will send the message `Msg` to the process whose pid is `Pid`. Message sending is an asynchronous operation and its evaluation result is the evaluated value of the sent message.

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:

```
receive
  Message1 [when Guard1] ->
    Action1 ;
  Message2 [when Guard2] ->
    Action2 ;
  ...
  MessageN [when GuardN] ->
    ActionN
end
```

Figure 2.3: Erlang receive block

In the Erlang code pattern given in Figure 2.3, `receive` and `end` are primitives that denote the scope of the receive block. A 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.

Generally speaking, the order in which messages appear in the mailbox is not necessarily the same as the order those messages were sent because messages may be concurrently sent from parallel threads or distributed nodes. Nevertheless, messages sent from the same sender to the same receiver is guaranteed to appear in the mailbox in the order they were sent, if both are delivered.

The `echo_actor` example, given in Figure 2.4, spawns two processes, both of which verbatim print their received messages. The `print` function terminates as soon as the first message is processed. On the contrary, `loop` is a recursive function that can always process new messages. Line 34 of Figure 2.4 confirms two properties of message sending 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 the `start` function (i.e. `hello4`), 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",
               [Val+1]),
17           counter(Val+1)
18     end.
19
20 send_msgs(_, 0) -> true;
21 send_msgs(S, Count) ->
22     S ! "Hello",
23     send_msgs(S, Count-1).
24
25 %% Terminal Output:
26 %% 1> c(counter).
27 %% {ok,counter}
28 %% 2> counter:start().
29 %% increase counter to 1
30 %% <0.95.0>
31 %% 2 message(s) has/have been processed
32 %% 3 message(s) has/have been processed
33 %% 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 is probably the most important concept in the OTP design principles [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. The structure of a supervision tree may look like the one presented in Figure 2.6, where supervisors are represented by squares and workers are represented by circles. The example is cited from [Ericsson AB., 2013c, Section 1.1]. Restart strategies of each supervisor (Section 2.3.2.2), however, are removed from the figure since they are not related to the central ideas discussed at this moment.

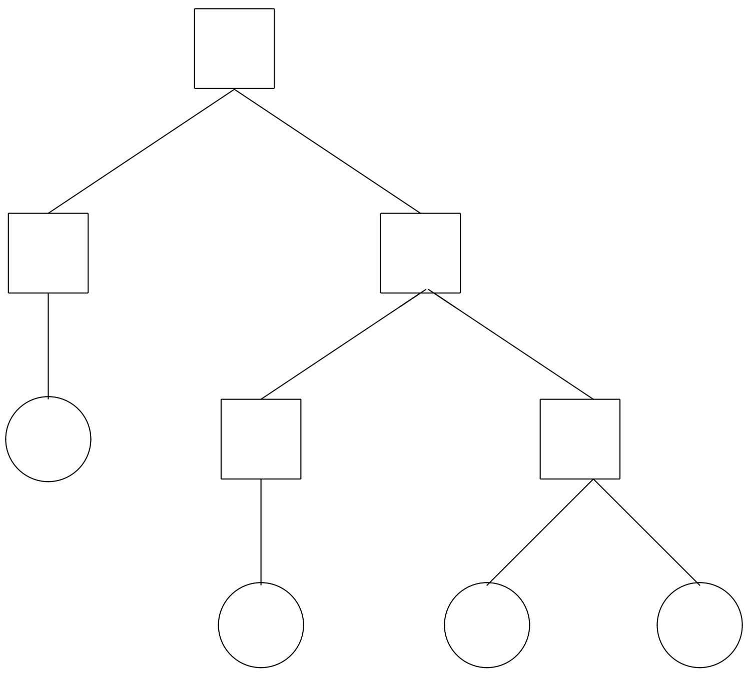


Figure 2.6: A Supervision Tree

2.3.2.1 An Erlang Supervision Example

We present the code for a simple Erlang supervisor in Figure 2.7. The core computation of the `supervision_demo` example is a problematic function `loop`, which eventually will try to compute the quotient of 10 divided by 0. As the test result shows, the problematic process has been restarted twice when it raises an error. The process is killed when it has failed for the third time within 60 seconds.

The short example implements the `supervisor` behaviour and specifies its supervision policy in its `init/1` method. The supervision policy reads as the following: spawn a worker process by calling `spawn(supervisor_demo, start, [Count])`; always restart the child when it fails if it does not fail more than twice within 60 seconds; if the child process fails more frequently than allowed, terminate it immediately. Alternative Erlang supervision strategies are explained in the following.

2.3.2.2 Supervision Strategy

In principle, a supervisor is accountable for starting, stopping and monitoring its child processes according to the policy specified in its `init/1` method, according to the API given in Figure 2.8 [Ericsson AB., 2013c]. A supervision strategy contains two parts: a restart strategy applies to all children and a list of child specifications for each child.

An Erlang supervisor employs one of the four restart strategies which specify its behaviour when one of its child fails. A supervisor with `one_for_one` strategy restart a child when it fails. A supervisor with `one_for_all` strategy restart all children when one of them fails. A supervisor with `rest_for_one` strategy restart the failed child and other children that started later than the failed child, according to their order in the list of child specifications. A supervisor with `simple_one_for_one` strategy is a `one_for_one` supervisor whose children are dynamically added instances of the same process.

A child specification 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.

```
-module(supervisor_demo).
-behaviour(supervisor).

-export([start/0, start/1, loop/1, init/1]).

start() ->
    supervisor:start_link(supervisor_demo, [2]).

start(Count) ->
  io:fwrite("Starting...~n"),
  Pid=spawn_link(supervisor_demo, loop, [Count]),
  {ok, Pid}.

init([Count]) ->
  {ok, {{one_for_one, 2, 60},
         [{supervisor_demo, {supervisor_demo, start, [Count]},
         permanent, brutal_kill, worker, [supervisor_demo]}]}}.

loop(Count) ->
  io:fwrite("~w / ~w is ~w ~n", [10, Count, 10/Count]),
  loop(Count-1).

%% 1> c(supervisor_demo).
%% {ok,supervisor_demo}
%% 2> supervisor_demo:start().
%% Starting...
%% 10 / 2 is 5.0
%% 10 / 1 is 10.0
%% <0.39.0>
%% Starting...
%% 10 / 2 is 5.0
%% 10 / 1 is 10.0
%% Starting...
%% 3>
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.40.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% 10 / 2 is 5.0
%% 3>
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.42.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% 10 / 1 is 10.0
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.43.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% ** exception error: shutdown
```

Figure 2.7: Erlang Example: Supervision Demo

```
Module:init(Args) -> Result

Args = term()
Result = {ok, {{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
  RestartStrategy = strategy()
  MaxR = integer()>=0
  MaxT = integer()>0
  ChildSpec = child_spec()

child_spec() =
   {Id :: child_id(),
    StartFunc :: mfargs(),
    Restart :: restart(),
    Shutdown :: shutdown(),
    Type :: worker(),
    Modules :: modules()}

child_id() = term()

mfargs() =
   {M :: module(), F :: atom(), A :: [term()] | undefined}

restart() = permanent | transient | temporary

shutdown() = brutal_kill | timeout()

strategy() = one_for_all
         | one_for_one
         | rest_for_one
         | simple_one_for_one

worker() = worker | supervisor

modules() = [module()] | dynamic
```

Figure 2.8: Erlang API: supervision

2.3.3 Other 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 Java and Scala programming.

2.3.3.1 The Behaviour Principle

A Behaviour in Erlang is similar to an interface, a trait, or an abstract class in the objected oriented programming. It implements 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, is coded by implementing 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.

2.3.3.2 The Application Principle

A software system on the OTP platform is made of a group of components called applications. To define an application, users implements two callback functions of the `application` behaviour: `start/2` and `stop/1`. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process, registered as `application_controller`.

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, registered as `dist_ac`, 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. Second, 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 a given time, and nodes that *may* be started in a given time.

2.3.3.3 The Release Principle and The Release Handling Principle

A complete Erlang system consists of one or more applications, packaged in a release resource file. Different version of a release can be upgraded or downgraded at run-time dynamically by calling APIs in the `release_handler` module in the SASL (System Architecture Support Libraries) application. Hot swapping on an entire release application is a distinct feature of Erlang/OTP, which aims at design and running non-stop applications.

2.3.3.4 Appling OTP Design Principles in Java and Scala

To sum, we make an analogy between Erlang/OTP design principles and common practices in Java and Scala programming, summarised in Table 2.1.

OTP Design Principle	Java/Scala Analogy
Behaviour	defining an abstract class, an interface, or a trait.
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 and Scala Programming

First, the notion of callback functions in Erlang/OTP is close to to the notion of abstract methods in Java and Scala. An OTP behaviour that only defines the signature of callback functions can be ported to Java and Scala as an interface. An OTP behaviour that implements some behaviour functions can be ported as an abstract class to *prevent* multiple inheritance, or a trait to *permit* multiple inheritance. Since Java does not have the notion of trait, porting an Erlang/OTP module that implements multiple behaviours requires a certain amount of refactoring work.

Second, since the Erlang application module is just a special behaviour, we can define an equivalent interface `Application` which contains two abstract methods: `start` and `stop`. To mimic the dynamic type system of Erlang system, the `start` method may be declared as `public static void start(String name, Object... arguments)` and as `def start(name:String, arguments:Any*):Unit` in Java and Scala respectively.

Third, Erlang releases correspond to packages in Java and Scala but hot code swapping is not directly supported by JVM. During the development of the TAkka library, we noticed that hot code swapping on a key component can be mimicked by updating the reference to that component.

The final OTP design principle, Supervision, has no direct correspondence in Java and Scala programming practices. The next section introduces the Akka library which implements the supervision principle.

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, 2007b; 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])

Although many Akka designs have their origin in Erlang, the Akka Team at Typesafe Inc. devises a set of connected concepts that explains Actor programming in the Akka framework. This subsection begins with a short Akka example, followed by elaborate explanations of involved concepts.

```
package sample.akka

import akka.actor.{Actor, ActorRef, ActorSystem, Props}

class StringCounter extends Actor {
  var counter = 0;
  def receive = {
    case m:String =>
      counter = counter +1
      println("received "+counter+" message(s):\n\t"+m)
  }
}

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

object StringCounterTest extends App {
  val system = ActorSystem("StringCounterTest")
  val counter = system.actorOf(Props[StringCounter], "counter")

  val handler = system.actorOf(Props[MessageHandler])
  system.eventStream.subscribe(handler,classOf[akka.actor.UnhandledMessage]);
  counter ! "Hello World"
  counter ! 1
  val counterRef =
      system.actorFor("akka://StringCounterTest/user/counter")
  counterRef ! "Hello World Again"
  counterRef ! 2
}


/*
Terminal output:
received 1 message(s):
  Hello World
received 2 message(s):
  Hello World Again
unhandled message:1
unhandled message:2
*/
```

Figure 2.9: Akka Example: A String Counter

The code presented in Figure 2.9 defines and uses an actor which counts String messages it receives. An Akka actor implements its message handler by defining a `receive` method of type `PartialFunction[Any, Unit]`. In the `StringCounterTest` application, we create an Actor System (Section 2.4.1.1), initialise an actor (Section 2.4.1.2) inside the Actor System by passing a corresponding `Props` (Section 2.4.1.4), and send messages to the created actor via its actor references (Section 2.4.1.5). Unexpected messages to the `counter` actor (e.g. line 28 and 31) are handled by an instance of `MessageHandler`, a helper actor for the test application. Lastly, the order in which the four output messages are printed is non-deterministic, but "Hello World" is always printed before "Hello World Again" and "unhandled message:1" is always printed before "unhandled message:2".

2.4.1.1 Actor System

In Akka, every actor is resident in an Actor System. An actor system organises related actors in a tree structure and provides services such as thread scheduling, network connection, and logging. One or several local and remote actor systems consist a complete application.

To create an actor system, users provide a name and an optional configuration to the `ActorSystem` constructor. For example, an actor system is created in Figure 2.9 by the following code.

```
val system = ActorSystem("StringCounterTest")
```

In the above, an actor system of name StringCounterTest is created at the machine where the program runs. The above created actor system uses the default Akka system configuration which provides a simple logging service, a round-robin style message router, but does not support remote messages. Customized configuration can be encapsulated in a `Config` instance and passed to the `ActorSystem` constructor, or specified as part of the application configuration file. This short tutorial will not look into customized configurations, which have minor differences in different Akka versions, and are not related to our central topics.

2.4.1.2 The Actor Class

An Akka Actor has four groups of fields given in Figure 2.10: *i*) its *state*, *ii*) its *behaviour* functions, *iii*) an `ActorContext` instance encapsulating its contextual information, and *iv*) the *supervisor strategy* for its children. This subsection explains the *state* and *behaviour* of actors, which are required when defining an Actor class. Overriding default *actor context* and *supervisor strategy* will be explained in later subsections.

```
trait Actor extends AnyRef
  type Receive = PartialFunction[Any, Unit]

  abstract def receive: Actor.Receive
  implicit final val self: ActorRef
  implicit val context: ActorContext
  def supervisorStrategy: SupervisorStrategy

  final def sender: ActorRef

  def preStart(): Unit
  def preRestart(reason: Throwable, message: Option[Any]): Unit
  def postRestart(reason: Throwable): Unit
  def postStop(): Unit}
```

Figure 2.10: Akka API: Actorr

An Akka actor may contain some mutable variables and immutable values that represent its *internal state*. Each Akka actor has an actor reference, `self`, to which messages can be sent to that actor. The value of `self` is initialised when the actor is created. Notice that `self` is declared as a value field (`val`), rather than a variable field (`var`), so that its value cannot be changed. In addition to *immutable states*, sometimes *mutable states* are also required. For example, Akka developers believe that the sender of the last message shall be recorded and easily fetched by calling the `sender` method. In the `StringCounter` example, we straightforwardly add a `counter` variable which is initialized to 0 and is incremented each time when a String message is processed.

There are two drawbacks of using mutable internal variables to represent states. Firstly, those variables will be reset each time when the actor is restarted, either due to a failure caused by itself or be enforced by its supervisor for other reasons. Secondly, mutable internal variables result in the difficulty of implementing a consistent cluster environment where actors may be replicated to increase reliability [Kuhn et al., 2012]. The alternatives of working with mutable states will be discussed in Section 3.10.

There are two kinds of `behaviour` functions of an actor. The first type of behaviour functions is a `receive` function which defines its action to incoming messages. The `receive` function is declared as an `abstract` function, which must be implemented otherwise the class cannot be initialised. The second group of behaviour functions has four overridable functions which are triggered before the actor is started (`preStart`), before the actor is restarted (`preRestart`), after the actor is restarted (`postRestart`), and when the actor is permanently terminated (`postStop`). The default implementation of those four functions take no action when they are invoked.

Look closely at the `receive` function of the `StringCounter` actor in Figure 2.9, it actually has type `Function[String, Unit]` rather than the declared type `PartialFunction[Any, Unit]`. The definition of `StringCounter` is accepted by the Scala compiler because `PartialFunction` does not check the completeness of the input patterns. The behaviour of processing non-String messages, however, is undefined in the `receive` method.

2.4.1.3 Message 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 behaviour function of an Akka actor must be able to process the message it is given. If the message does not match any message pattern of the current behaviour, 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. Line 24 of the String Counter example demonstrates how to subscribe a type of messages in the event stream of an actor system.

2.4.1.4 Actor Creation with Props

An instance of the `Props` class, which perhaps strands for "properties", specifies the configuration of creating an actor. A `Props` instance is immutable so that it can be consistently shared between threads and distributed nodes.

Figure 2.11 gives part of the APIs of the `Props` class and its companion object. The `Props` is defined as a *final class* so that users cannot define subclasses of it. Moreover, users are not encouraged to initialise a `Props` instance by directly using its constructor. Instead, a `Props` should be initialised by using one of the apply methods supplied by the Props object. From the perspective of software design patterns, the Props object is a *Factory* for creating instances of the Props class.

```scala
package akka.actor
final case class Props(deploy: Deploy, clazz: Class[_],
                       args: Seq[Any]) extends Product with Serializable

object Props extends Serializable
  def apply[T <: Actor]()(implicit arg0: ClassManifest[T]): Props
  def apply(clazz: Class[_], args: Any*): Props
```

Figure 2.11: Akka API: Props

We have seen an example of creating a Props instance in Figure 2.9, that is:

```scala
  Props[StringCounter]
```

which is short for

```scala
  Props.apply[StringCounter]()(implicitly[ClassManifest[StringCounter]])
```

The APIs of the first Props.apply method is carefully designed to take the advantages of the Scala language. Firstly, the word apply can be omitted when it is used as a method name. Secondly, round brackets can be omitted when a method does not take any argument. Thirdly, implicit parameters are automatically provided if implicit values of the right types can be found in scope. As a result, in most cases, only the class name of an Actor is required when creating a Props of that actor.

Alternatively, calling the second apply method requires a *class object* and arguments sending to the class constructor. For example, the above Props can be alternatively created by the following code:

```scala
  Props(classisOf[StringCounter])
```

In the above, the predefined function classOf[T] returns a class object for type T. More arguments can be sent to the constructor of StringCounter if there is one that 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.

Once an instance of Props is created, an actor can be created by passing that Props instance to the actorOf method of ActorSystem or ActorContext.

In Figure 2.9, we have seen that `system.actorOf` creates an actor directly supervised by the system guidance actor for all user-created actors (`user`). Calling `context.actorOf` creates an actor supervised by the actor represented by that context. Details of actor context and supervision will be given in Section 2.4.1.6 and Section 2.4.2 respectively.

2.4.1.5 Actor Reference and Actor Path

Actors collaborate by sending messages to each others via actor references of message receivers. An actor reference has type `ActorRef`, which provides a `!` method to which messages are sent. For example, in the `StringCounter` example in Figure 2.9, `counter` is an actor reference to which the message `"Hello world"` is sent by the following code:

```
counter ! "Hello world"
```

which is the syntactic sugar for

```
counter.!("Hello world") .
```

```
abstract class ActorRef extends Comparable[ActorRef] with Serializable

abstract def path: ActorPath
def !(message: Any)(implicit sender: ActorRef = Actor.noSender): Unit
final def compareTo(other: ActorRef): Int
final def equals(that: Any): Boolean
def forward(message: Any)(implicit context: ActorContext): Unit
```

Figure 2.12: Akka API: Actor Reference

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 an actor name, which shall not be conflict with its siblings, to the address of its parent. Examples of akka addresses are:

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

The first address represents the path to a local actor. Inspired by the syntax of uniform resource identifier (URI), an actor address consists of a 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 Actor Context

The `ActorContext` class has been mention a few times in previous sections. This section explains what the contextual information of an Akka actor includes, with a reference to the following APIs cited from [Typesafe Inc. (a), 2012].

The API in Figure 2.13 shows two groups of methods: those for interacting with other actors (line 3 to line 24), and those for controlling the behaviour of the represented actor (line 26 to line 30).

As mentioned in Section 2.4.1.4, calling the `context.actorOf` method creates a child actor supervised by the actor represented by that context. Every actor has a name distinguished from its siblings. If a user assigned name is conflict with the name of another existed actor, an `InvalidActorNameException` raises. If the user does not provide a name when creating an actor, a system generated name will be used instead. The return value of the `actorOf` method is an actor reference pointing to the created actor.

Once an actor is created, its actor reference can be obtained by inquiring on its actor path using the `actorFor` method. Since version 2.1, Akka encourages obtaining actor references via a new method `actorSelection`, whose return

```scala
package akka.actor
trait ActorContext
  abstract def actorOf(props: Props, name: String): ActorRef
  abstract def actorOf(props: Props): ActorRef

  abstract def child(name: String): Option[ActorRef]
  abstract def children: Iterable[ActorRef]
  abstract def parent: ActorRef

  abstract def props: Props
  abstract def self: ActorRef
  abstract def sender: ActorRef

  implicit abstract def system: ActorSystem

  def actorFor(path: Iterable[String]): ActorRef
  def actorFor(path: String): ActorRef
  def actorFor(path: ActorPath): ActorRef
  def actorSelection(path: String): ActorSelection

  abstract def watch(subject: ActorRef): ActorRef
  abstract def unwatch(subject: ActorRef): ActorRef

  abstract def stop(actor: ActorRef): Unit

  abstract def become(behavior: Receive,
                      discardOld: Boolean = true): Unit
  abstract def unbecome(): Unit
  abstract def receiveTimeout: Duration
  abstract def setReceiveTimeout(timeout: Duration): Unit
```

Figure 2.13: Akka API: Actor Context

value broadcasts messages it receives to all actors in its subtrees. The `actorFor` method is deprecated in version 2.2. Code in this thesis still uses the deprecated `actorFor` method because, in most cases, our simple examples only need to send message to a particular actor.

Actor context is also used to fetch some states inside the actor. For example, the context of an actor records references to its parent and children, the props used to create that actor, actor references to itself and the sender of the last message, and the actor system where the actor is resident.

Ported from the Erlang design, using the `watch` method, an Akka actor can monitor the liveness of another actor, which is not necessarily its child. The liveness monitoring can be cancelled by calling the `unwatch` method. Another method ported from Erlang is the `stop` method which sends a termination signal to an actor. Since supervision is obligatory in Akka and users are encouraged to managing the lifecycle of an actor either inside the actor or via its supervisor, we believe that those three methods are redundant in Akka. For all examples studied in this thesis, there is no client application that requires any of those three methods.

Finally, actor context manages two behaviours of the actor it represents. The first behaviour specified by the actor context is the timeout within which a new message shall be received or a `ReceiveTimeout` message is sent to the actor. The second behaviour managed by the actor context is the handler for incoming messages. The next subsection explains how to hot swap the message handler of an actor using the `become` and `unbecome` method.

2.4.1.7 Hot Swapping on Message Handler

In the `StringCounter` example given at the beginning of this section, a message handler is defined in the `receive` method. The `StringCounter` is a simple actor which only requires an initial message handler that never changes. In some other cases, it is required to update the message handler of an actor at runtime. For example, an online calculator may upgrade to a version that supports more types of calculation.

Message handlers of an Akka actor are kept in a stack of its `context`. A message handler is pushed to the stack when the `context.become` method is called; and is popped out from the stack when the `context.unbecome` method is called. The message handler of an actor is reset to the initial one, i.e. the `receive` method, when it is restarted.

To demonstrate hot swapping on the behaviour of Akka actors, Figure 2.14 defines a calculator. The calculator starts with a basic version that can only compute multiplication. When it receives an `Upgrade` command, it upgrades to an advanced version that can compute both multiplication and division. The advanced calculator downgrades to to the basic version when it receives a `Downgrade` command. For simplicity, the demo code does not consider the potential *division by zero* problem, an error that can be tolerated if the actor is properly supervised.

```scala
package sample.akka
import akka.actor._
case object Upgrade
case object Downgrade
case class Mul(m:Int, n:Int)
case class Div(m:Int, n:Int)
class CalculatorServer extends Actor {
  import context._
  def receive = simpleCalculator
  def simpleCalculator:Receive = {
    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
    case Upgrade =>
      println("Upgrade")
      become(advancedCalculator, discardOld=false)
    case op =>    println("Unrecognised operation: "+op)
  }
  def advancedCalculator:Receive = {
    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
    case Div(m:Int, n:Int) =>  println(m +" / "+ n +" = "+ (m/n))
    case Downgrade =>
      println("Downgrade")
      unbecome();
    case op =>    println("Unrecognised operation: "+op)
  }
}
object CalculatorUpgrade extends App {
    val system = ActorSystem("CalculatorSystem")
    val calculator:ActorRef = system.actorOf(Props[CalculatorServer],
        "calculator")
   calculator ! Mul(5, 1)
   calculator ! Div(10, 1)
   calculator ! Upgrade
   calculator ! Mul(5, 2)
   calculator ! Div(10, 2)
   calculator ! Downgrade
   calculator ! Mul(5, 3)
   calculator ! Div(10, 3)
}
/*  Terminal output:
5 * 1 = 5
Unrecognised operation: Div(10,1)
Upgrade
5 * 2 = 10
10 / 2 = 5
Downgrade
5 * 3 = 15
Unrecognised operation: Div(10,3)
 */
```

Figure 2.14: Akka Behaviour Swap Example

2.4.2 Supervision in Akka

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

A distinguishing feature of the Akka library is making supervision obligatory by restricting the way of creating actors. Recall that every user-created actor is initialised in one of the two ways: using the `system.actorOf` method so that it is a child of the system guardian actor; or using the context.actorOf method so that it is 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 `context.stop(child)`, where `child` is an actor reference.

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 is not implemented in Akka because Akka actor does not specify the order of 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.

Figure 2.15 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.

```
package akka.actor
abstract class SupervisorStrategy
case class OneForOne(restart:Int, time:Duration)(decider: Throwable =>
  Directive) extends SupervisorStrategy with Product with Serializable
case class OneForAll(restart:Int, time:Duration)(decider: Throwable =>
  Directive) extends SupervisorStrategy with Product with Serializable

sealed trait Directive extends AnyRef
object Escalate extends Directive with Product with Serializable
object Restart extends Directive with Product with Serializable
object Resume extends Directive with Product with Serializable
object Stop extends Directive with Product with Serializable
```

Figure 2.15: Akka API: Supervisor Strategies

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 Case Study: A Fault-Tolerant Calculator

Figure 2.16 defines a simple calculator which supports multiplication and division. The simple calculator does not consider the problematic case of dividing a number by 0, in which case 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.

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

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

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

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

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

/*
Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
10 / 5 = 2
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
3 * 2 = 6
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
*/
```


Figure 2.16: Supervised Calculator

The last message is not processed because the both calculators are terminated because the simple calculator fails more frequently than allowed.

2.5 The Scala Type System

One of the key design principles of the TAkka library, described in the subsequent Chapters, is using type checking to detect some errors at the earliest opportunity. Since both TAkka and Akka are built using the Scala programming language [Odersky et al., 2004; Odersky., 2013], this section summarises key features of the Scala type system that benefit the implementation of the TAkka library.

2.5.1 Parameterized Types

A *parameterized type* $\mathtt{T[U_1,...,U_n]}$ consists of a type constructor `T` and a positive number of type parameters $\mathtt{U_1,...,U_n}$ [Odersky., 2013]. The type constructor `T` must be a valid type name whereas a type parameter $\mathtt{U}_i$ can either be a specific type value or a type variable. Scala Parameterized Types are similar to Java and C# generics and C++ templates, but express *variance* and *bounds* differently as explained later.

2.5.1.1 Generic Programming

To demonstrate how to use Scala parameterized types to do generic programming, Figure 2.17 gives a simple stack library and an associated client application ported from a Java example found in [Naftalin and Wadler, 2006, Example 5-2]. The example defines an abstract data type Stack, an implementation class `ArrayStack`, a utility method `reverse`, and client application `Client`.

In the example, `Stack` is defined as a *trait*, which is an analogy to an abstract class that supports multiple inheritance. The `Stack` trait defines the signature of three methods: `empty`, `push`, and `pop`. A Stack maintains a collection of data to which an entity can be added (the *push* operation) or be removed (the *pop* operation) in a *Last-In-First-Out* order. The `empty` method defined in the `Stack` trait returns `true` if the collection does not contain any data. The Stack trait takes a type parameter `E` which appears in the `push` and `pop` methods as well. The argument of the `push` method has type `E` so that only data of type `E` can be added to the `Stack`. Consequently, the `pop` method is expected to return data of type `E`.

```
trait Stack[E] {
  def empty():Boolean
  def push(elt:E):Unit
  def pop():E
}
```

```
import scala.collection.mutable.ArrayBuffer
class ArrayStack[E] extends Stack[E]{
  private val list:ArrayBuffer[E] = new ArrayBuffer[E]()
  def empty():Boolean = { return list.size == 0 }

  def push(elt:E):Unit = { list += elt }
  def pop():E = {
    val elt:E = list.remove(list.size-1)
    return elt
  }

  override def toString():String = {
    return "stack"+list.toString.drop(11)
  }
}
```

```
object Stacks {
  def reverse[T](in:Stack[T]):Stack[T] = {
    val out = new ArrayStack[T]
    while(!in.empty){
      val elt = in.pop
      out.push(elt)
    }
    return out
  }
}
```

```
object Client extends App {
  val stack:Stack[Integer] = new ArrayStack[Integer]
  var i = 0
  for(i <- 0 until 4) stack.push(i)
  assert(stack.toString().equals("stack(0, 1, 2, 3)"))
  val top = stack.pop
  assert(top == 3 && stack.toString().equals("stack(0, 1, 2)"))
  val reverse = Stacks.reverse(stack)
  assert(stack.empty)
  assert(reverse.toString().equals("stack(2, 1, 0)"))
}
```

Figure 2.17: Scala Example: A Generic Stack Library

The `ArrayStack` class implements the `Stack` trait and overrides the `toString` method which gives a string representation of the `Stack`. An `ArrayStack` instance internally saves data in an `ArrayBuffer`. Prepending an element to an `ArrayBuffer` (line 8) takes constant time while removing an element from an `ArrayBuffer` takes time linear regarding to the buffer size.

The utility method `reverse` repeatedly pops data from one stack and pushes it onto the stack to be returned. Different to Java, Scala classes do not have static members. Therefore the `reverse` method is defined in a *singleton object*, the only instance of a class with the same name. Notice that, the object `Stacks` is not type-parameterized, but its method `reverse` is.

The `Client` application creates an empty stack of integers, pushes four integers to it, pops out the last one, and then saves the remainder into a new stack in reverse order. The code `stack.push(i)` takes an advantage of the Scala compiler called *autoboxing*, which converts a primitive type `Int` to its corresponding object wrapper class `Integer`. The example code using autoboxing to write cleaner code.

2.5.1.2 Type Bounds

In the above section, we defined a type parameterized stack to which only values whose type is the same as its type variable can be pushed. The benefit is that data popped from the type parameterized stack always has expected type. In a sense, the `push(elt:E):Unit` method of `Stack[E]` specified in Figure 2.17 is overly restrictive because it only accepts an argument of type `E`, but not data of a subtype of `E`.

Figure 2.18 gives a more flexible Stack, the types of whose elements are either the same as the type parameter or subtypes of the type parameter. In Figure 2.18, the signature of `push` is changed to `push[T<:E](elt:T):Unit`, with an additional type parameter `T<:E` which denotes that `T` is a subtype of `E`. In Scala, `E` is called the *upper bound* of `T`. Similarly, `T>:E` means `T` is a supertype of `E` and `E` is called the *lower bound* of `T`. In Scala, `Any` is the supertype of all types and `Nothing` is the subtype of all types.

The remaining code in Figure 2.18 is the same as the code in Figure 2.17 except that, on line 4 of the `Client` example, the value of `i` need to be explicitly converted to an `Integer` because the current Scala compiler is not sophisticated enough.

```scala
trait Stack[E] {
  def empty():Boolean
  def push[T <:  E](elt:T):Unit
  def pop():E
}
```

```scala
import scala.collection.mutable.ArrayBuffer
class ArrayStack[E] extends Stack[E]{
  private val list:ArrayBuffer[E] = new ArrayBuffer[E]()
  def empty():Boolean = {
    return list.size == 0
  }
  def push[T <:  E](elt:T):Unit = {
    list += elt
  }
  def pop():E = {
    val elt:E = list.remove(list.size-1)
    return elt
  }
  override def toString():String = {
    return "stack"+list.toString.drop(11)
  }
}
```

```scala
object Stacks {
  def reverse[T](in:Stack[T]):Stack[T] = {
    val out = new ArrayStack[T]
    while(!in.empty){
      val elt = in.pop
      out.push(elt)
    }
    return out
  }
}
```

```scala
object Client extends App {
  val stack:Stack[Integer] = new ArrayStack[Integer]
  var i = 0
  for(i <- 0 until 4) stack.push(new Integer(i))
  assert(stack.toString().equals("stack(0, 1, 2, 3)"))
  val top = stack.pop
  assert(top == 3 && stack.toString().equals("stack(0, 1, 2)"))
  val reverse = Stacks.reverse(stack)
  assert(stack.empty)
  assert(reverse.toString().equals("stack(2, 1, 0)"))
}
```

Figure 2.18: Scala Example: A Generic Stack Library using Type Bounds

2.5.1.3 Variance Under Inheritance

An important issue that is intentionally skirted in Section 2.5.1.1 is how variance under inheritance works in Scala. Specifically, if $\mathtt{T}_{sub}$ is a subtype of `T`, is `Stack[`$\mathtt{T}_{sub}$`]` the subtype of `Stack[T]`, or reversely? Unlike Java generic collections [Naftalin and Wadler, 2006], which are always invariant on the type parameter, Scala users can explicitly specify one of the three types of variance as part of the type declaration using variance annotation as summarised in Table 2.2, paraphrased from [Wampler and Payne, 2009, Table 12.1].

Variance Annotation	Description
+	Covariant subclassing. i.e. $X[T_{sub}]$ is a subtype of $X[T]$, if T_{sub} is a subtype of T.
-	Contravariant subclassing. i.e. $X[T^{sup}]$ is a subtype of $X[T]$, if T_{sup} is a supertype of T.
default	Invariant subclassing. i.e. cannot substitute $X[T^{sup}]$ or $X[T_{sub}]$ for $X[T]$, if T_{sub} is a subtype of T and T is a subtype of T_{sup}.

Table 2.2: Variance Under Inheritance

A variance annotation constrains positions where the annotated type variable may appear. Specifically, covariant, contravariant, and invariant type variables can only appear in covariant position, contravariant position, and invariant position respectively. The Scala compiler checks if types with variance are used consistently according to a set of rules given in [Odersky., 2013, Section 4.5]. As a programmer, the author of this thesis often find that it is easier to uses variant types according to a variant of the *Get and Put Principle*.

The *Get and Put Principle* for Java Generic Collections [Naftalin and Wadler, 2006, Section 2.4] read as the follows:

> **The Get and Put Principle:** *Use an extends wildcard when you only get values out of a structure, use a super wildcard when you only put values into a structure, and don't use a wildcard when you both get and put.*

When use generic types with variance in Scala, the general version is:

> **The General Get and Put Principle:** *Use a type in covariant positions when you only get values out of a structure, use a type in contravariant positions when you only put values into a structure, and use a type in invariant positions when you both get and put.*

Take the function type for example, a user *put*s an input value into its input channel, *get*s a return value from its output channel. According to the

```scala
trait Stack[+E] {
  def empty():Boolean
  def push[T>:E](elt: T): Stack[T]
  def pop():(E, Stack[E])
}
```

```scala
import scala.collection.immutable.List
class ArrayStack[E](protected val list:List[E]) extends Stack[E]{
  def empty():Boolean = { return list.size == 0 }
  def push[T >:E](elt: T): Stack[T] = { new ArrayStack(elt :: list) }
  def pop():(E, Stack[E]) = {
    if (!empty) (list.head, new ArrayStack(list.tail))
    else throw new NoSuchElementException("pop of empty stack")
  }
  override def toString():String = {
    return "stack"+list.toString.drop(4)
  }
}
```

```scala
object Stacks {
  def reverse[T](in:Stack[T]):Stack[T] = {
    var temp = in
    var out:Stack[T] = new ArrayStack[T](Nil)
    while(!temp.empty){
      val eltStack = temp.pop
      temp = eltStack._2
      out = out.push(eltStack._1)
    }
    return out
  }
}
```

```scala
object Client extends App {
  var stack:Stack[Integer] = new ArrayStack[Integer](Nil)
  var i = 0
  for(i <- 0 until 4) { stack = stack.push(i) }
  assert(stack.toString().equals("stack(3, 2, 1, 0)"))
  stack.pop match {
    case (top, stack) =>
      assert(top == 3 && stack.toString().equals("stack(2, 1, 0)"))
      val reverse:Stack[Integer] = Stacks.reverse(stack)
      assert(reverse.toString().equals("stack(0, 1, 2)"))
      val anystack:Stack[Any] = reverse.push(3.0)
      assert(anystack.toString().equals("stack(3.0, 0, 1, 2)"))
  }
}
```

Figure 2.19: Scala Example: A Covariant Immutable Stack

General Get and Put Principle, a function is contravariant in the input type and covariant in the output type.

This section concludes with an immutable Stack that is covariant on its type parameter, as shown in Figure 2.19. A stack is covariant on its type parameter because, for example, a stack that saves a collection of `Integer` values is also a stack that saves a collection of `Any` values. However, if the type of Stack is declared as `Stack[+E]`, the signature of its `push` method *cannot* be

```
def push[T<:E](elt:T):Unit
```

while its `pop` method always returns a value of type `E`; otherwise, a user can put a value of any type to a stack of integer. The trick is, as shown in the code, making the `Stack[+E]` class an *immutable* collection whose `push` and `pop` methods do not modify its content but return a new stack.

2.5.2 Scala Type Descriptors

As in Java, generic types are erased by the Scala compiler. To record type information that is required at runtime but might be erased, Scala users can ask the compiler to keep the type information by using the `Manifest` class.

The Scala standard library contains four manifest classes as shown in Figure 2.20. A `Manifest[T]` encapsulates the runtime type representation of some type `T`. `Manifest[T]` a subtype of `ClassManifest[T]`, which declares methods for subtype (<:<) test and supertest (>:>). The object `NoManifest` represents type information that is required by a paraterized type but is not available in scope. `OptManifest[+T]` is the supertype of `ClassManifest[T]` and `OptManifest`.

```
package scala.reflect
trait OptManifest[+T] extends Serializable

object NoManifest extends OptManifest[Nothing] with Serializable

trait ClassManifest[T] extends OptManifest[T] with Serializable
  def <:<(that: ClassManifest[_]): Boolean
  def >:>(that: ClassManifest[_]): Boolean
  erasure: Class[_]

trait Manifest[T] extends ClassManifest[T] with Serializable
```

Figure 2.20: Scala API: Manifest Type Hierarchy

```scala
package sample.other

import scala.reflect._

object ManifestExample extends App {
  assert(List(1,2.0,"3").isInstanceOf[List[String]])
  // Compiler Warning :non-variable type argument String in type
      List[String] is unchecked since it is eliminated by
  // erasure

  case class Foo[A](a: A)
  type F = Foo[_]
  assert(classManifest[F].toString.equals(
          "sample.other.ManifestExample$Foo[<?>]"))
  assert(NoManifest.toString.equals("<?>"))

  assert(manifest[List[Int]].toString.equals(
          "scala.collection.immutable.List[Int]"))
  assert(manifest[List[Int]].erasure.toString.equals(
          "class scala.collection.immutable.List"))

  def typeName[T](x: T)(implicit m: Manifest[T]): String = {
    m.toString
  }
  assert(typeName(2).equals("Int"))

  def boundTypeName[T:Manifest](x: T):String = {
    manifest[T].toString
  }
  assert(boundTypeName(2).equals("Int"))

  def isSubType[T: Manifest, U: Manifest] = manifest[T] <:< manifest[U]
  assert(isSubType[List[String], List[AnyRef]])
  assert(! isSubType[List[String], List[Int]])
}
```

Figure 2.21: Scala ExampleI: Manifest Example

The code example in Figure 2.21 shows common usages of Manifest. There are three ways of obtaining a manifest: using the Methods `manifest` (line 16) or `classManifest` (line 12), using an implicit parameter of type `Manifest[T]` (line 21), or using a `context bound` of a type parameter (line 26). Context bound can be seen as a syntactic sugar for implicit parameters without a user-specified parameter name. The `isSubType` method defined at line 31 tests if the first `Manifest` represents a type that is a subtype of the typed represented by the second `Manifest`.

2.6 Summing Up

To review, the Actor Model [Hewitt et al., 1973] is proposed for designing concurrent systems. It is employed by Erlang [Armstrong, 2007b] and other programming languages. Erlang developers designed the Supervision Principle in 1999 when the Erlang/OTP library is released as an open-source project. With the supervision principle, actors are supervised by their supervisors, who are responsible for initializing and monitoring their children. Erlang developers claimed that applications that using the supervision principle have achieved a high availability [Armstrong, 2002]. Recently, the actor programming model and the supervision principle have been ported to Akka, an Actor library written in Scala. Although Scala is a statically typed language and provides a sophistical type system, the type of messages sent to Akka actors are dynamically checked when they are processed. The next chapter presents the design and implementation of the TAkka library where type checks are involved in the earliest opportunity to expose type errors.

Chapter 3

TAkka: Design and Implementation

(This chapter is expanded from [TAKKA, Chapter X])

In the last Chapter, we have seen actor programming and supervision in Erlang/OTP and Akka. Erlang/OTP is written in Erlang, a dynamically typed language, whereas Akka is written in Scala, a statically typed language. A key advantages of static typing is that it detects some type errors at an early stage, i.e. at compile time. Nevertheless, messages sent to Akka actors are dynamically typed.

The key claim in this chapter is that actors in supervision trees can be typed by parameterizing the actor class with the type of messages that it expects to receive. This chapter presents the design of the TAkka library. We outline how static and dynamic type checking are used to prevent ill-typed messages. We show that supervision tree can be constructed for type-parameterized actors in the same ways as for actors without type parameter. This chapter concludes with a brief discussion about design alternatives used by other actor libraries

3.1 TAkka Example: a String Counter

We begin with an illustrative TAkka example in Figure 3.1. The example is ported from the string counter example given in Figure 2.9. The TAkka code is similar to its Akka equivalence, with a few differences marked in blue colour.

The first difference is that the `TypedActor` class in TAkka takes a type parameter that indicates the type of expected messages. In our example, `StringCounter` is an actor which only expects String messages. Consequently, its `typedReceive` function has a function type `String => Unit`. The type is not explicitly declared in code because it can be inferred and checked by the Scala

```
package sample.takka

import takka.actor.{TypedActor, ActorRef, ActorSystem, Props}

class StringCounter extends TypedActor[String] {
  var counter = 0;
  def typedReceive = {
    case m =>
      counter = counter +1
      println("received "+counter+" message(s):\n\t"+m)
  }
}

object StringCounterTest extends App {
  val system = ActorSystem("StringCounterTest")
  val counter = system.actorOf(Props[String, StringCounter], "counter")

  counter ! "Hello World"
//  counter ! 1
// type mismatch; found : Int(1) required: String
  val counterString =
      system.actorFor[String]("akka://StringCounterTest/user/counter")
  counterString ! "Hello World Again"
  val counterInt =
      system.actorFor[Int]("akka://StringCounterTest/user/counter")
  println("Hello")
  counterInt ! 2
}

/*
Terminal Output:

received 1 message(s):
  Hello World
received 2 message(s):
  Hello World Again
Exception in thread "main" java.lang.Exception:
    ActorRef[akka://StringCounterTest/user/counter] does not exist or
    does not have type ActorRef[Int]
 */
```

Figure 3.1: TAkka Example: A String Counter

compiler. In an Eclipse IDE with Scala plug-in, the following type information is shown on screen when mouseover the `typedReceive` method:

```
def typedReceive: String => Unit
```

For the same reason, the type of `m`, which is `String`, is omitted as well.

Secondly, the type of messages sending to an actor reference is statically checked. In the TAkka version of the `StringCounter` example, the compiler infers that the type of `counter`, declared at line 16, has type `ActorRef[String]`, to which only String messages can be sent. Sending a non-String message, however, results in a compile error.

Thirdly, dynamic type checking is involved at the earliest opportunity when static type checking is impossible or inadequate. For example, when a user looks up an actor reference by its type and path, as at line 21 and 23, the compiler cannot check if there will be an actor of a compatible type at that path when the program is executed. Although the type error at line 23 is not statically detected, a run-time exception raises as soon as that line is executed, much earlier than the time when the ill-typed actor reference is used.

Because sending an actors a message of unexpected type is prevented, there is no need to define a handler for unexpected messages in our TAkka example. Eliminating ill-typed messages benefit both users and developers of actor-based services. For users, since messages are transmitted asynchronously, it is easier to trace the source of potential errors if they are captured earlier, especially in distributed environments. For service developers, they can focus on the logic of the services rather than worrying about incoming messages of unexpected types.

3.2 Type-parameterized Actor

A TAkka actor has type `TypedActor[M]`. It inherits the Akka `Actor` trait to minimize implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor APIs. Instead, programmers are encouraged to use the typed fields given in Figure 3.2. Unlike other actor libraries, every TAkka actor class takes a type parameter `M` which specifies the type of messages expected by the actor. The same type parameter is used as the input type of the `typedReceive` function. The actor reference pointing to itself, `typedSelf`, has type `ActorRef[M]` to which only messages of type `M` can be sent. The actor context for the actor, `typedContext`, has type `ActorContext[M]`. On the other hand, fields not directly related to messages processing has the same type signature as their Akka equivalent.

```
package takka.actor

abstract class TypedActor[M:Manifest] extends akka.actor.Actor
  protected def typedReceive:Function[M, Unit]

  def typedReceive:M=>Unit
  val typedSelf:ActorRef[M]
  val typedContext:ActorContext[M]
  var supervisorStrategy: SupervisorStrategy

  def preStart(): Unit
  def preRestart(reason: Throwable, message: Option[M]): Unit
  def postRestart(reason: Throwable): Unit
  def postStop(): Unit
```

Figure 3.2: TAkka API: TypedActor

Notice that the type of `typedReceive` is `Function[M, Unit]`, whereas the type of `receive` in Akka Actor is `PartialFunction[Any, Unit]`. There are two distinctions between `Function` and `PartialFunction`. The advantage of `PartialFunction` is that users can define a new function that merges the input domains of two other partial functions. The advantage of `Function` is that the completeness of pattern matching can be checked by the compiler if the input type of the function is a sealed trait or a sealed class, whose subclasses must be defined in the same file. Figure 3.3 gives a Scala example that illustrate the above features. The last function, `fruitNameF`, shows that the syntactically shorter definition of `fruitnamePF` can be equivalently defined. Moreover, because the `typedReceive` function should not be visible outside the Actor class, the advantage of using `PartialFunction` is not clear. On the other hand, completeness check of message patterns might be a useful feature in practice.

The two immutable fields of `TypedActor`: `typedContext` and `typedSelf`, are automatically initialized when the actor is created. Library users may override the default supervisor strategy in the way explained in Section 3.9. The implementation of the `typedReceive` method, on the other hand, is always provided by users.

Notice that `takka.actor.TypedActor` inherits `akka.actor.Actor`. A critical problem of using inheritance is that, ironically, dynamically typed Akka APIs, which are against in this thesis, are still available to TAkka users. Unfor-

```scala
sealed trait Fruit
case object Apple extends Fruit
case object Orange extends Fruit

object PartialFunctionExample extends App {
  val appleNamePF:PartialFunction[Fruit, Unit] = {
    case Apple =>
      println("Apple")
  }
  val orangeNamePF:PartialFunction[Fruit, Unit] = {
    case Orange =>
      println("Orange")
  }
  val fruitnamePF: PartialFunction[Fruit, Unit] = appleNamePF orElse
      orangeNamePF
  assert(fruitnamePF(Orange).equals("Orange"))

  val appleNameF:Fruit=>Unit ={
    case Apple =>
      println("Apple")
  }
//compiler warning: match may not be exhaustive. It would fail on the
    following input: Orange
  val orangeNameF:Fruit=>Unit ={
    case Orange =>
      println("Orange")
  }
//compiler warning: match may not be exhaustive. It would fail on the
    following input: Apple
  val fruitNameF:Fruit=>Unit ={
    case Apple =>
      appleNameF(Apple)
    case Orange =>
      orangeNameF(Orange)
  }
}
```

Figure 3.3: Scala Example: PartialFunction and Function

tunately, this limitation cannot be overcome by using delegation because, as we have seen in Akka APIs, a child actor is created by calling the `actorOf` method from its supervisor's actor context, which cannot accessed outside the supervisor. `TypedActor` is the only TAkka class that is implemented using inheritance. All other TAkka classes and traits are either implemented by delegating tasks to Akka counterparts or rewritten in TAkka. We believe that re-implementing the TAkka Actor library requires a similar amount of work for implementing the Akka Actor library.

3.3 Type -parameterized Actor Reference

The last section explains the type-parameterised Actor class, `TypedActor[M]`, whose message handler only considers messages of the expected type `M`. Such a design only works in system which either provides a reasonable handler for undefined messages at the receiver side, or is able to prevent ill-typed messages at the sender side. As mentioned in Section 2.4.1.3, undefined messages are handled differently in Erlang and different Akka versions. Each mechanism has its own rationality. Unfortunately, there is no known single mechanism that meets the requirements of all applications. The Akka development team tends to provide more ways to handle unexpected messages at the receiver side. In contrast, the TAkka library set an ambitious goal to prevent ill-typed messages at the sender side. We achieve the goal by adding a type parameter to the `ActorRef` class.

The API of `ActorRef` is given in Figure 3.4. The constructor of the `ActorRef` class takes two parameters: one type parameter that indicates the type of expected message and one implicit arguments that records the `Manifest` of the type parameter. In most cases, users obtain actor references via API calls rather than creating actor references themselves. The type of a TAkka actor reference, however, only has a type parameter.

Like in Erlang and Akka, users send a message to an actor reference via its `!` method. Sending an actor a message of a different type causes an error at compile time. By using type-parameterized actor references, the receiver does not need to worry about unexpected messages, while senders can be sure that messages will be understood and processed, as long as the message is delivered.

An actor usually can react to a finite set of different message patterns whereas our notion of actor reference only takes one type parameter. In a type system that supports untagged union types, no special extension is required. In a type system which supports polymorphism, `ActorRef` should be contravariant on its type argument `M`, denoted as `ActorRef[-M]` in Scala. To understand why `ActorRef` is contravariant, let's consider the *Get and Put Principle* and an illustrative example. `ActorRef` is contravariant because users only `put` values to an actor reference but never get values out of it. The illustrative example considered here is a simple calculator defined in Figure 3.5. The calculator defined in the example can compute the result of two types of operations: multiplication and division. Hence, `Multiplication` is a subtype of `Operation`. It is clear that `ActorRef[Operation]` is a subtype of `ActorRef[Division]` because if users can send both multiplication requests and division requests to an actor reference, they can send multiplication requests only to that actor reference.

```
package takka.actor

abstract class ActorRef[-M](implicit mt:Manifest[M])
  private val untypedRef:akka.actor.ActorRef

  def !(message: M):Unit
  def publishAs[SubM<:M](implicit smt:Manifest[SubM]):ActorRef[SubM]

  abstract def path: akka.actor.ActorPath
  final def compareTo(other: ActorRef[_]): Int
  final def equals(that: Any): Boolean
  // no forward method
```

Figure 3.4: TAkka API: Actor Reference

For the ease of use, `ActorRef` provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method has three advantages. Firstly, the semantics of `publishAs` does not require a deep understanding of underlying concepts like contravariance and inheritance. Secondly, explicit type conversion using `publishAs` is type checked by the compiler which ensures that the type of the result is a supertype of the input actor reference. 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. The last advantage is important in Scala because a library developer may not have access to code written by others.

```
package sample.takka

import takka.actor.ActorRef
import takka.actor.ActorSystem
import takka.actor.Props
import takka.actor.TypedActor

sealed trait Operation
case class Multiplication(m:Int, n:Int) extends Operation
case class Division(m:Int, n:Int) extends Operation

class Calculator extends TypedActor[Operation] {
  def typedReceive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m, n) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}

object SupervisorTest extends App{
  val system = ActorSystem("MySystem")
  val calculator:ActorRef[Operation] = system.actorOf(Props[Operation,
      Calculator], "calculator")
  val multiplicator = calculator.publishAs[Multiplication]

  calculator ! Multiplication(3, 2)
  multiplicator ! Multiplication(3, 3)
//  multiplicator ! Division(6, 2)
  //Compiler Error: type mismatch; found : sample.takka.Division
      required:
//   sample.takka.Multiplication
}

/*
Terminal Output:
3 * 2 = 6
3 * 3 = 9
*/
```

Figure 3.5: TAkka Example: Contravariant Actor Reference

3.4 Type Parameterized Props

A instance of type `Props[M]` is used when creating an actor of type `TypedActor[M]`. A Prop of type `Prop[M]` can be created by one of the two factory methods provided by the `Props` object.

```
package takka.actor

final case class Props[-T](props:  akka.actor.Props)

object Props
  def apply[T, A<:TypedActor[T]] (implicit arg0:Manifest[A]): Props[T]
  def apply[T](clazz:  Class[_ <:  TypedActor[T]],args:Any*):  Props[T]
```

Figure 3.6: TAkka API: Props

In Section 3.1, a Props for creating an instance of `StringCounter` is created by the following code

```
  Props[String, StringCounter]
```

In the above short example, the compiler checks that `StringCounter` is a subtype of `TypedActor[String]`, and provides a value for the implicit parameter, which has type `Manifest[StringCounter]`.

The TAkka `Props` class is contravariant on its type parameter because users can create an actor by providing a props that is able to create actors that can handle more types of messages.

3.5 Type Parameterized Actor Context

An actor context describes the contextual information of an actor. Because each actor is an independent computational primitive, an actor context is private to its corresponding actor. By using APIs in Figure 3.7, an actor can (*i*) create a child actor supervised by itself, (*ii*) fetch some of its states, (*iii*) retrieve an actor reference corresponding to a given actor path and type using the `actorFor` method, (*iv*) set a timeout denoting the time within which a new message must be received using the `setReceiveTimeout` method, and (*v*) update its behaviours using the `become` method. Compared with corresponding Akka APIs, TAkka methods take an additional type parameter whose meaning will be explained shortly.

```
package takka.actor

abstract class ActorContext[M:Manifest]
  def actorOf [Msg] (props: Props[Msg])
                    (implicit mt:  Manifest[Msg]): ActorRef[Msg]
  def actorOf [Msg] (props: Props[Msg], name: String)
                    (implicit mt:  Manifest[Msg]): ActorRef[Msg]

  def remoteActorOf[Msg](props:Props[Msg])
                    (implicit mt:Manifest[Msg]):ActorRef[Msg]
  def remoteActorOf[Msg](props:Props[Msg], name:String)
                    (implicit mt:Manifest[Msg]) :ActorRef[Msg]

  // no child, children, and parent

  def props:Props[M]
  lazy val typedSelf:ActorRef[M]
  // no sender

  implicit def system : ActorSystem

  def actorFor[Msg] (actorPath: String)
      (implicit mt:  Manifest[Msg]):  ActorRef[Msg]
  def actorFor[Msg](actorPath:
akka.actor.ActorPath)(implicit mt:Manifest[Msg]):ActorRef[Msg]
  // no actorSelection

  // no watch, unwatch, and stop

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

  // no unbecome

  def receiveTimeout : Duration
  def setReceiveTimeout (timeout: Duration): Unit
```

Figure 3.7: TAkka API: Actor Context

An actor creates a child actor using the `actorOf` method or the `remoteActorOf` method. If no user-specified name is provided for the child, a system-generated one will be used. The `actorOf` method returns a TAkka actor reference which internally maintains a type descriptor and an Akka actor reference. Somehow, the Akka actor reference returned by the Akka system cannot be used remotely because its actor path does not include IP and port number. The `remoteActorOf` method is implemented in TAkka as a complement to `actorOf`. The `remoteActorOf` returns an actor reference that can be used remotely if the actor system enables remote communication, otherwise it raises an Exception. Calling `remoteActorOf` takes longer time than calling `actorOf` as IP and port information need to be fetched from the system configuration.

The contextual information of an actor includes the `Props` used to create that actor, the typed actor reference pointing to that actor, and the actor system where the actor is resident in. TAkka removes APIs that inquiry on the actor reference of of parent and children for two reasons. Firstly, the type of parent and children of an actor varies from one actor to another. Secondly, actor references to parent and children of an actor can be obtained using the `actorFor` method if their paths and types are known by the user. A TAkka actor context does not record the value of `sender` neither because its type changes at run time. We recommend the Erlang-style message pattern, in which the actor reference to the message sender is part of the message if a reply message is expected.

The two `actorFor` methods fetches an actor reference of the expected type located at an actor path. The task is actually delegated to the actor system, as explained in Section 3.8

♠**Todo: complete this chapter** ♠

set a timeout denoting the time within which a new message must be received using the `setReceiveTimeout` method, and

update its behaviours using the `become` method.

3.6 Backward Compatible Hot Swapping

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

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

The `become` method is implemented as in Figure 3.8. 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.

```scala
package takka.actor

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

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

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

    new ActorRef[SupM] {
      val untypedRef = untypedContext.self
    }
  }
}

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

Figure 3.8: Hot Swapping in TAkka

```
package sample.takka

import takka.actor._

sealed trait Op
sealed trait Command extends Op
case class Mul(m:Int, n:Int) extends Command
case object Upgrade extends Command
case class Div(m:Int, n:Int) extends Op

class CalculatorServer extends TypedActor[Command] {
  def typedReceive = {
    case Mul(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Upgrade =>
      println("Upgrading ...")
      typedContext.become(advancedCalculator)
  }
  def advancedCalculator:Op=>Unit = {
    case Mul(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Div(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
    case Upgrade =>
      println("Upgraded.")
  }
}
object CalculatorUpgrade extends App {
    val system = ActorSystem("CalculatorSystem")
    val simpleCal:ActorRef[Command] = system.actorOf(Props[Command,
CalculatorServer], "calculator")
   simpleCal ! Mul(5, 1)
   simpleCal ! Upgrade

   val advancedCal =
system.actorFor[Op]("akka://CalculatorSystem/user/calculator")
   advancedCal ! Mul(5, 3)
   advancedCal ! Div(10, 3)
   advancedCal ! Upgrade
}
/* Terminal Output:
5 * 1 = 5
Upgrading ...
5 * 3 = 15
10 / 3 = 3
Upgraded.
 */
```

Figure 3.9: TAkka Behaviour Upgrade Example

3.7 Typed Name Server

A key advantages of static typing is that it detects some type errors at an early stage, i.e. at compile time. The TAkka library is designed to detect type errors as early as possible. Nevertheless, not all type errors can be statically detected, and some dynamic type checks are required. To address this issue, a notion of run-time type descriptor is required. This section summarizes the type reflection mechanism in Scala and explains how it benefits the implementation of our typed name server. Our typed name server can be straightforwardly ported to other platforms that support type reflection.

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.

3.7.1 Typed Name and Typed Value

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 **??**:

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

case class TValue[T:Manifest](val value:T){
 private [takka] val t:Manifest[_] = manifest[T]
}
```

Figure 3.10: TSymbol and TValue

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

3.7.2 Operations

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

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

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

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

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

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

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

Notice that `unset` and `get` operations succeed as long as the associated type of the input name is the supertype of the associated type of the registered name. To permit polymorphism, the `hashcode` method of `TSymbol` defined in Figure **??** 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.

♠an example. one page code, half page explanation ♠

3.7.3 Alternative Type Comparison Methods

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.

♠explain with example. one page code, half page explanation ♠

3.8 Look up Typed Actor References

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

3.9 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.

```
package takka.actor

case class NotRemoteSystemException(system:ActorSystem) extends
    Exception("ActorSystem: "+system+" does not support remoting")

abstract class ActorSystem
    val system:akka.actor.ActorSystem
    def stop(actorRef:ActorRef[_])
    def deadLetters : ActorRef[Any]
    def isTerminated: Boolean

    def actorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
    def actorOf[Msg:Manifest](props:Props[Msg],
        aname:String):ActorRef[Msg]
    def remoteActorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
    def remoteActorOf[Msg:Manifest](props:Props[Msg],
        aname:String):ActorRef[Msg]

    def actorFor[M:Manifest](actorPath: String): ActorRef[M]
```

Figure 3.11: Akka API: Actor System

Figure **??** 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 **??** 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.

♠example code: safe calculator again ♠

3.9.1 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

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

```scala
object ActorSystem {
  def apply(sysname: String, config: Config, classLoader: ClassLoader): ActorSystem = new ActorSystem {
     val system = akka.actor.ActorSystem(sysname, config, classLoader)
    system.actorOf(akka.actor.Props(new ActorTypeChecker), "ActorTypeChecker")
  }
}

abstract class ActorSystem {

  val system:akka.actor.ActorSystem

  def actorOf[Msg:Manifest](props:Props[Msg], aname:String):ActorRef[Msg] = {
    val actor = new ActorRef[Msg] { val untypedRef = system.actorOf(props.props, aname) }
    NameServer.set(TSymbol[ActorRef[Msg]](Symbol(actor.path.toString())), actor)
    actor
  }

  private def actorFor[M:Manifest](actorPath: akka.actor.ActorPath): ActorRef[M]= {
    // short description
  }


  private class ActorTypeChecker extends akka.actor.Actor{
    def receive = {
      case Check(path, t) =>
        NameServer.get(TSymbol(Symbol(path.toString))(t) ) match {
          case None => sender ! NonCompatible // not registered actor or incompatible type
          case Some(_) => sender ! Compatible
        }
    }
  }
}
```

Figure 3.12: Actor System: actorFor

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

Figure 3.13: Supervisor Strategies

given as follows:

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

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

- `Kill`

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

- `PoisonPill`

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

- `Restart`

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

- `ReceiveTimeout`

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

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

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

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

3.10 Design Alternatives

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

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

```scala
package sample.akka;

import akka.actor.ActorSystem
import akka.actor.Props
import akka.actor.TypedActor
import akka.actor.TypedProps
import akka.actor.UnhandledMessage

trait StringCounterTypedActor{
  def processString(m:String)
}

class StringCounterTypedActorImpl (val name:String) extends
    StringCounterTypedActor{
  private var counter = 0;
  def this() = this("default")

  def processString(m:String) {
    counter = counter +1
    println("received "+counter+" message(s):\n\t"+m)
  }
}

object StringCounterTypedActorTest extends App {
  val system = ActorSystem("StringCounterTest")
  val counter:StringCounterTypedActor =
      TypedActor(system).typedActorOf(TypedProps[StringCounterTypedActorImpl](),
      "counter")
  counter.processString("Hello World")

  val handler = system.actorOf(Props(new MessageHandler()))
  system.eventStream.subscribe(handler,classOf[akka.actor.UnhandledMessage]);

//  counter ! "Hello World"
//  Compiler Error:
//   value ! is not a member of sample.akka.StringCounterTypedActor
  val counterRef =
      system.actorFor("akka://StringCounterTest/user/counter")
  counterRef ! "Hello World Again"
  counterRef ! 2
}
/*  Terminal Output:

received 1 message(s):
  Hello World
unhandled message:Hello World Again
unhandled message:2
 */
```

Figure 3.14: Akka Example: String Counter using TypedActor

Chapter 4

Evolution, Not Revolution

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

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

In previous sections, we have seen how to use Akka actors in an Akka system (Figure **??**) and how to use TAkka actors in a TAkka system (Figure 3.1). In the following, we will explain how to use TAkka actors in an Akka system and how to use an Akka actor in a TAkka system.

4.1 Akka actor in Akka system

4.2 TAkka actor in TAkka system

4.3 TAkka actor in Akka system

It is often the case that an actor-based library is implemented by one organization but used in a client application implemented by another organization. If a developer decided to upgrade the library implementation using TAkka actors, for example, upgrading the Socko Web Server Imtarnasan and Bolton [2012], the Gatling Excilys Group [2012] stress testing tool, or the core library of the Play framework Typesafe Inc. (c) [2013] as we have done in Section 5.2, how would the upgrade affects client code, especially legacy applications built using the Akka library? TAkka actor and actor reference are implemented using inheritance and delegation respectively so that no changes are required for legacy applications.

TAkka actors inherits Akka actors. In Figure **??**, the actor implementation is upgraded to the TAkka version as in Figure 3.1. 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 **??**, 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.

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

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

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

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

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

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

  val untypedserver = takkaserver.untypedRef

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

  untypedserver ! "Hello TAkka"
  untypedserver ! 4
}

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

Figure 4.1: TAkka actor in Akka application

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

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

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

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

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

Figure 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 TAkka library. We show that the Wadler's type pollution problem can be avoided in a straightforward way by using TAkka. We further assess the TAkka library by porting examples written in Erlang and Akka. Results show that TAkka detects type errors without 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].

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

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

Figure 5.1: Template for Model-View-Controller

5.2 Expressiveness and Correctness

Table **??** lists the examples used for expressiveness and correctness. We selected examples from Erlang Quiviq Arts et al. [2006] and open source Akka projects to ensure that the main requirements for actor programming are not unintentionally neglected. Examples from Erlang Quiviq are re-implemented using both Akka and TAkka. Examples from Akka projects are re-implemented using TAkka. Following the suggestion in Hennessy and Patterson [2006], we assess the overall code modification and code size by calculating the geometric mean of all examples. The evaluation results in Table 5.2 show that when porting an Akka program to TAkka, about 7.4% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because TAkka code does not need to handle unexpected messages. On average, the total program size of Akka and TAkka applications are almost the same.

A type error is reported by the compiler when porting the Socko example Imtarnasan and Bolton [2012] from its Akka implementation to equivalent TAkka implementation. SOCKO is a library for building event-driven web services. The SOCKO designer defines a `SockoEvent` class to be the supertype of all events. One subtype of `SockoEvent` is `HttpRequestEvent`, representing events generated when an HTTP request is received. The designer further implements subclasses of `Method`, whose `unapply` method intends to pattern match `SockoEvent` to `HttpRequestEvent`. The SOCKO designer made a type error in the method declaration so that the `unapply` method pattern matches `SockoEvent` to `SockoEvent`. The type error is not exposed in test examples because the those examples always passes instances of `HttpRequestEvent` to the `unapply` method and send the returned values to an actor that accepts messages of `HttpRequestEvent` type. Fortunately, the design flaw is exposed when upgrading the SOCKO implementation using TAkka.

5.3 Efficiency, Throughput, and Scalability

The TAkka library is built on top of Akka so that code for shared features can be re-used. The three main source of overheads in the TAkka implementation are: (i) the cost of adding an additional operation layer on top of Akka code, (ii) the cost of constructing type 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 3.1. When n ranges from 10^4 to 10^5, the TAkka implementation is about 2 times slower as the Akka implementation. The cost of the last factor is close to the cost of transmitting the string representation of fully qualified type names.

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

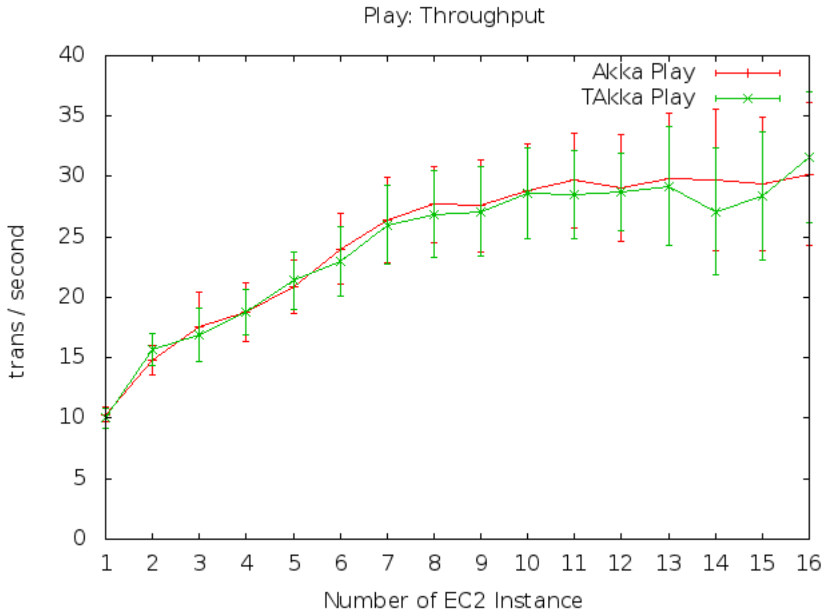
We further investigated the speed-up of multi-nodes TAkka applications by porting 6 micro benchmark examples, listed in Table **??**, from the BenchErl benchmarks in the RELEASE project Boudeville et al. [2012]. Each BenchErl benchmark spawns one master process and many child processes for a tested task. 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, TAkka and Akka implementations have almost identical run-time and scalability.

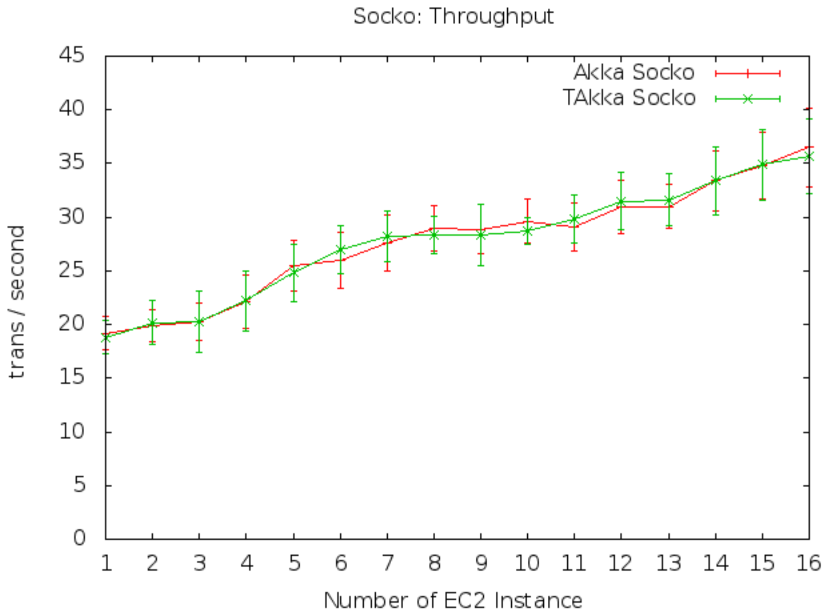
5.4 Assessing System Reliability

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 [2002] and the wide range of Akka use cases, how could software developers assure the reliability of their newly implemented applications?

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

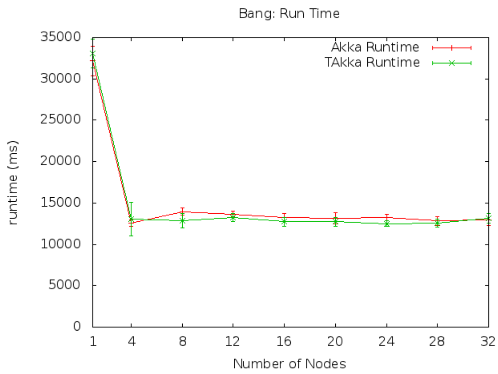


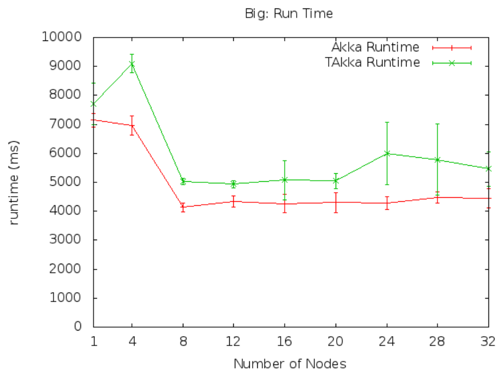
(a)



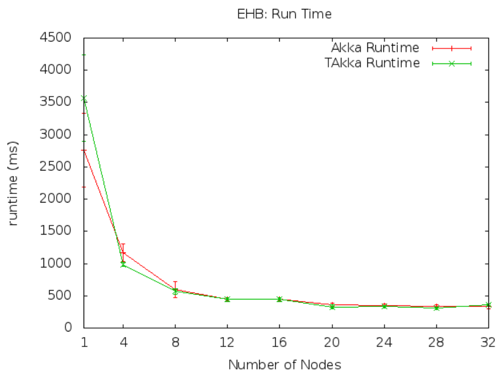
(b)

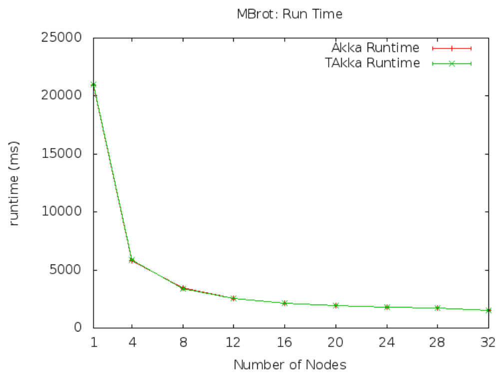
Figure 5.2: Throughput Benchmarks



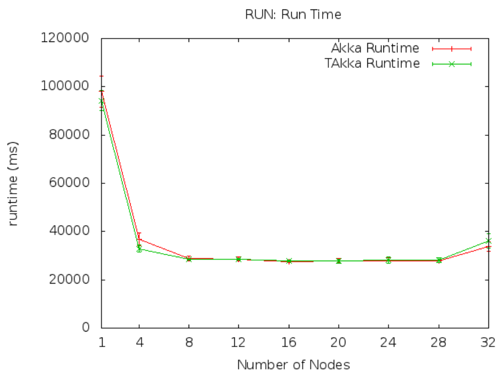


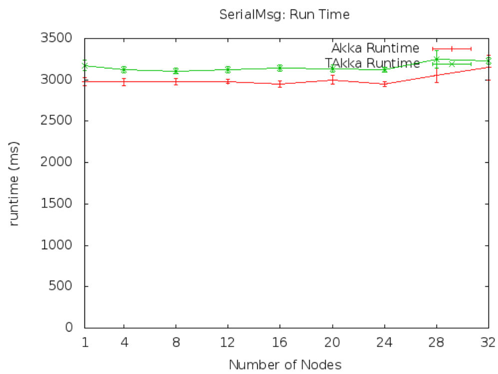
(a) (b)





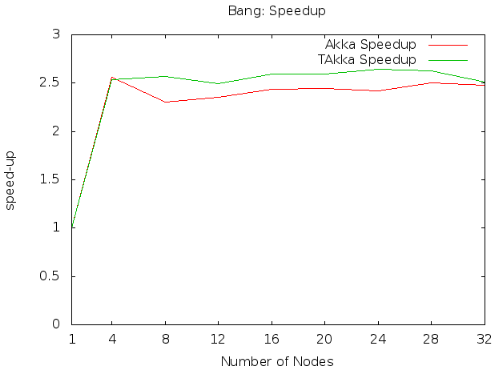
(c) (d)



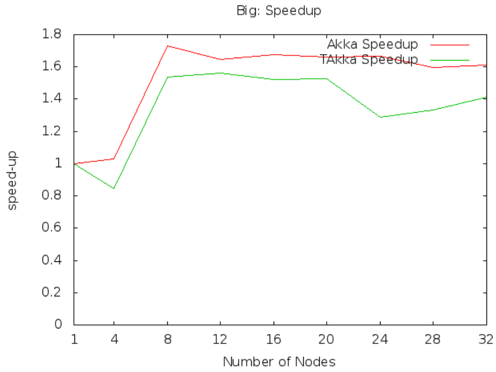


(e) (f)

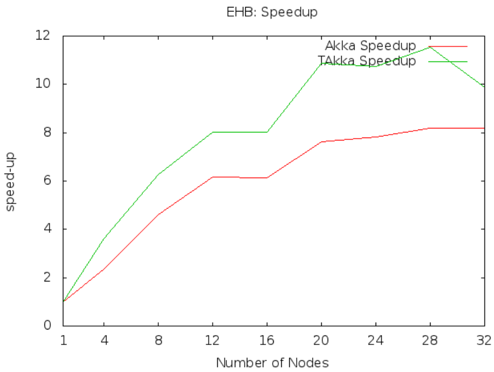
Figure 5.3: Runtime Benchmarks



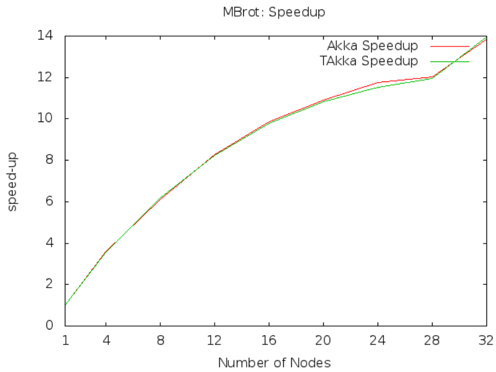
(a)



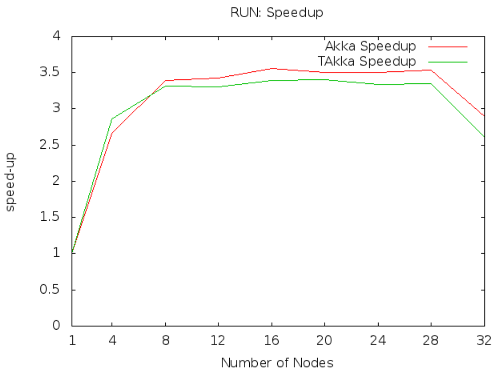
(b)



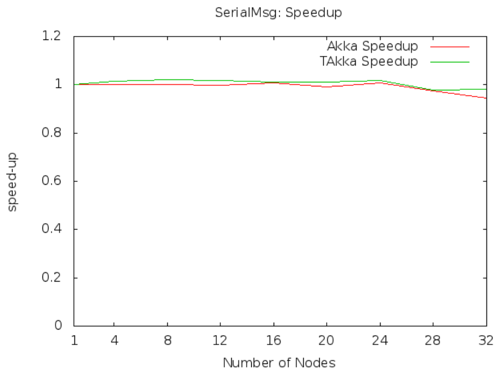
(c)



(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 (Section5.4.4) 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 **??**.

Figure **??** gives the API and the core implemention 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 excute a corresponding potentially problematic code as described in Table **??**. `PoisonPill` and `Kill` are handled by `systemMessageHandler` and can be overrided as described in Section 3.9.1. `ChaosException` and `ChaosNonTerminate`, on the other hand, are handled by the TAkka library and cannot be overrided.

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

Table 5.1: Examples for Correctness and Expressiveness Evaluation

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

Table 5.2: Results of Correctness and Expressiveness Evaluation

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

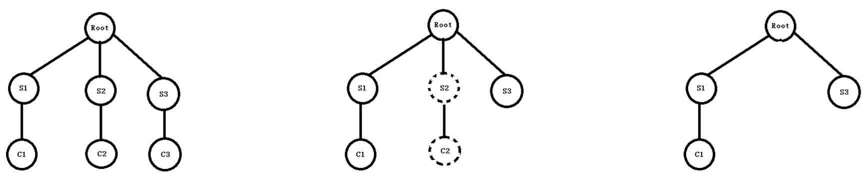
Table 5.4: TAkka Chaos Monkey Modes

```scala
class ChaosMonkey(victims:List[ActorRef[_]], exceptions:List[Exception]){
  private var status:Status = OFF;

  def setMode(mode:ChaosMode);
  def enableDebug();
  def disableDebug();
  def start(interval:FiniteDuration) = status match {
    case ON =>
throw new Exception("ChaosMonkey is running: turn it off before restart it.")
    case OFF =>
      status = ON
      scala.concurrent.future {
        repeat(interval)
      }
  }
  def turnOff()= {status = OFF}

  private def once() {
    var tempMode = mode
    if (tempMode == Random){
      tempMode = Random.shuffle(
                 ChaosMode.values.-(Random).toList).head
    }
    val victim = scala.util.Random.shuffle(victims).head
    tempMode match {
      case PoisonKill =>
        victim.untypedRef ! akka.actor.PoisonPill
      case Kill =>
        victim.untypedRef ! akka.actor.Kill
      case Exception =>
        val e = scala.util.Random.shuffle(exceptions).head
        victim.untypedRef ! ChaosException(e)
      case NonTerminate =>
        victim.untypedRef ! ChaosNonTerminate
    }
  }

  private def repeat(period:FiniteDuration):Unit = status match {
    case ON =>
      once
      Thread.sleep(period.toMillis)
      repeat(period)
    case OFF =>
  }
}

object ChaosMode extends Enumeration {
    type ChaosMode = Value
    val Random, PoisonKill, Kill, Exception, NonTerminate = Value
}
```


Figure 5.5: TAkka Chaos Monkey

```
sealed trait SupervisionViewMessage
case class SupervisionViewResponse(date:Date, reportorPath:ActorPath,
childrenPath:List[ActorPath]) extends SupervisionViewMessage
case class ReportViewTo(drawer:ActorRef[Map[Date,
    TreeSet[NodeRecord]]]) extends
 SupervisionViewMessage

case class SupervisionViewRequest(date:Date,
master:ActorRef[SupervisionViewResponse])
case class NodeRecord(receiveTime:Date, node:ActorPath,
childrenPath:List[ActorPath])

object ViewMaster{
  def apply(name:String, config: Config, topnodes:List[ActorRef[_]],
interval:FiniteDuration):ViewMaster

  def apply(name:String, topnodes:List[ActorRef[_]],
interval:FiniteDuration):ViewMaster

  def apply(topnodes:List[ActorRef[_]],
      interval:FiniteDuration):ViewMaster
}
```

Figure 5.6: Supersion View



(a) (b) (c)

Figure 5.7: Supervision View Example

Chapter 6

Future Work

Chapter 7

Conclusion

Bibliography

Agha, G. A. (1985). Actors: a model of concurrent computation in distributed systems.

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

Armstrong, J. (2007a). A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM.

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

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.

Baker, H. and Hewitt, C. (1977). Laws for communicating parallel processes.

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*.

Clinger, W. D. (1981). Foundations of actor semantics.

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.

Grief, I. (1975). Semantics of communicating parallel processes.

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. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc.

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

JActor Consulting Ltd (2013). JActor. `http://jactorconsulting.com/product/jactor/`. Accessed on Oct 2013.

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

Lieberman, H. (1981). Thinking about lots of things at once without getting confused: Parallelism in act 1.

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*. O'Reilly Media, Inc.

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

Odersky., M. (2013). The Scala Language Specication Version 2.8. Technical report, EPFL Lausanne, Switzerland.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the scala programming language. Technical report, Citeseer.

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.

Wampler, D. and Payne, A. (2009). *Programming Scala*. O'Reilly Series. O'Reilly Media.

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