Chapter 1

Introduction

Building reliable distributed applications is among the most difficult tasks facing programmers, and one which is becoming increasingly important due to the recent advent of web applications, cloud services, and mobile apps. Modern society relies on distributed applications which are executed on heterogeneous runtime environments, are tolerant of partial failures, and sometimes dynamically upgrade some of their components without affecting other parts.

A distributed application typically consists of components which handle some tasks independently, while collaborating on other tasks by exchanging messages. The robustness of a distributed application, therefore, can be improved by (i) using a fault-tolerant design to minimise the aftermath of partial failures, or (ii) employing type checking to detect some errors, including the logic of component implementations, and communications between components.

One of the most influential fault-tolerant designs is the supervision principle, proposed in the first release of the Erlang/OTP library in 1997 [Ericsson AB., 2013c]. The supervision principle states that concurrent components of an application should be encapsulated as actors, which make local decisions in response to received messages. Actors form a tree structure, where a parent node is responsible for monitoring its children and restarting them when necessary. The supervision principle is proposed to increase the robustness of applications written in Erlang, a dynamically typed programming language. Erlang application developers can employ the supervision principle by using related APIs from the Erlang/OTP library. It is reported that the supervision principle helped AXD301, an ATM (Asynchronous Transfer Mode) switch manufactured by Ericsson Telecom AB. for British Telecom, to achieve 99.9999999% (9 nines) uptime during a nine-month test [Armstrong, 2002]. Nevertheless, adopting the Supervision principle is optional in Erlang applications.

Aside from employing good design patterns, programmers can use typed programming languages to construct reliable and maintainable programs. Typed programming languages have the advantages of detecting some errors earlier, enforcing disciplined and modular programming, providing guarantees on language safety, and efficiency optimisation [Pierce, 2002].

Can programmers benefit from the advantages of both the supervision tree and type checking? In fact, attempts have been made in two directions: statically type checking Erlang programs and porting the supervision principle to statically typed systems.

Static checking in Erlang can be done via optional checking tools or rewriting applications using an Erlang variant that uses a statically typed system. Static analysis tools of Erlang include the Dialyzer [Ericsson AB., 2013a] and a fault tolerance analysis tool by Nyström [2009]. The Dialyzer tool is shipped with Erlang. It has identified a number of unnoticed errors in large Erlang applications that have been run for many years [Lindahl and Sagonas, 2004]. Nevertheless, the use of Dialyzer and other analysis tools is often involved in the later stages of Erlang applications development. In comparison with static analysis tools, simplified Erlang variants that use static type systems have been designed by Marlow and Wadler [1997], Sabelfeld and Mantel [2002], among others. As the expressiveness is often sacrificed in those simplified variants to some extent, code modifications are more or less required to make existing Erlang programs go through the type checker.

The second attempt is porting the notion of actors and supervision trees to statically typed languages, including Scala and Haskell. Scala actor libraries, including Scala Actors [Haller and Odersky, 2006, 2007] and Akka [Typesafe Inc. (a), 2012; Typesafe Inc. (b), 2012], use dynamically typed messages even though Scala is a statically typed language. Some recent actor libraries, including Cloud Haskell [Epstein et al., 2011], Lift [Typelevel ORG, 2013], and scalaz [WorldWide Conferencing, LLC, 2013], support both dynamically and statically typed messages, but do not support supervision. Can actors in supervision trees be statically typed?

The key claim in this thesis is that actors in supervision trees can be statically typed by parameterizing the actor class with the type of messages it expects to receive. Type-parameterized actors benefit both users and developers of actor-based services. For users, sending ill-typed messages is prevented at compile time. Because messages are usually transmitted asynchronously, it may be otherwise difficult to trace the source of errors at runtime, especially in distributed environments. For service developers, since unexpected messages are eliminated from the system, they can focus on the logic of the services rather than worrying about incoming messages of unexpected types.

Implementing type-parameterized actors in a statically-typed language, however, requires solving the following three problems:

1. A typed name server is required to retrieve actor references of specific types. A distributed system usually requires a name server which maps names of services to processes that implement that service. If processes are dynamically typed, the name server is usually implemented as a map from names to processes. Can a distributed name server maintain maps from the typed names and processes of corresponding types, and provide APIs for registering and fetching statically typed processes?

2. A supervisor actor must interact with child actors of different types. Each actor in a supervision tree needs to handle messages for both the purpose of supervision and its own specific interests. When all actors are parameterized by different types, is it practical to define a supervisor that communicates with children of different types?

3. Actors that receive messages from distinct parties may suffer from the type pollution problem, whereby a party imports too much type information about an actor and can send the actor messages not expected from it. Systems built on a layered architecture or the MVC model are often victims of the type pollution problem. As an actor receives messages from distinct parties using its sole channel, its type parameter is the union type of all expected message types. Therefore, unexpected messages can be sent to an actor which naively publishes its type parameter or permits dynamically typed messages. Can a type-parameterized actor publish itself as different types when it communicates with different parties?

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 supervised actors are parameterized by the type of messages they expect. The library (Chapter 3) mixes static and dynamic 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 actor specific communications. The decision is made so that type-parameterized actors of different types can form a supervision tree. In addition, the TAkka library is carefully designed so that Akka programs can gradually migrate to their TAkka equivalents (evolution) rather than requiring the provision of type parameters everywhere (revolution). Moreover, the type pollution problem can be straightforwardly avoided in TAkka.

- A framework for evaluating libraries that supports 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 with their Akka equivalents. Finally, we port the Chaos Monkey library and design a Supervision View library. The Chaos Monkey library tests whether exceptions are properly handled by supervisors. The Supervision View library dynamically captures the structure of supervision trees. We believe that similar evaluations can be done in Erlang and new libraries that support the supervision principle.

Chapter 2

Background and Related Work

This Chapter summarises work that influences the design and implementation of the TAkka library. It begins with a general introduction on the Actor programming model and the Supervision principle, then explains OTP design principles in Erlang, followed by a short tutorial on how to use the Actor programming model and the Supervision principle in the Akka library. The Chapter concludes with a summary of Scala features used in the TAkka implementation. The Actor model makes concurrent programming easy. The Supervision principle makes applications robust. The Supervision principle is introduced in the Erlang language. It becomes obligatory in the Akka library, which is implemented in the Scala language. Scala has a sophisticated type system, which enabled the experimental building of the more powerful and easier-to-use library, TAkka.

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 Grief [1975]. Baker and Hewitt [1977] later define a set of axiomatic laws for Actor systems. Other semantics of the Actor model include 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. Unlike the Erlang system, an Akka actor can be both a worker and a supervisor.

2.3 Erlang and OTP Design Principles

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].

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

The five OTP design principles are: The Behaviour Principle, The Application Principle, The Release Principle, The Release Handling Principle, and The Supervision Principle [Ericsson AB., 2013c]. The Supervision Principle was introduced in the previous section. This section describes the ideas 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 general Java and Scala programming practice.

2.3.1 The Behaviour Principle

A Behaviour in Erlang is similar to an interface, a trait, or an abstract class in object oriented programming. It defines 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 Erlang processes, including those in the Erlang standard library, are 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 makes code more maintainable and reliable. Standard Erlang/OTP behaviours include:

- *gen_server* for constructing the server of a client-server paradigm.
- *gen_fsm* for constructing finite state machines.
- *gen_event* for implementing event handling functionality.
- *supervisor* for implementing a supervisor in a supervision tree.

2.3.2 The Application Principle

A software system on the OTP platform is made up of a group of components called applications. To define an application, users implement 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 includes three parameters: the time for other nodes to start, nodes that *must* be started in a timeout, and nodes that *may* be started in a timeout.

2.3.3 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 versions 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 designing and running non-stop applications.

2.3.4 Applying OTP Design Principles in Java and Scala

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

First, the notion of callback functions in Erlang/OTP is close to that 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

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

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 whereas hot code swapping is not directly supported by JVM. During the development of the TAkka library, we noticed that dynamically upgrading key components can be mimicked by updating the references to those components.

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., 2013b]. 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.

The code presented in Figure 2.1 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 Scala, `Any` is the supertype of all types. The type `Unit` has a unique value. A method with the return type `Unit`, such as the `receive` method, represents a block of local actions. An analogy to such a method is a Java method which is declared `void`. 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 constitute 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 2.1 by the following code.

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

In the above, an actor system of name StringCounterTest is created in the machine where the program runs. The above created actor system uses the default Akka system configuration which provides a simple logging service, a

```scala
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 2.1: Akka Example: A String Counter

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 2.2: *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 2.2: Akka API: Actor

An Akka actor may contain some mutable variables and immutable values that represent its *internal state*. Each Akka actor has an actor reference, `self`, through 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 a String message is processed.

There are two drawbacks to 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 3.12.

There are two kinds of `behaviour` functions of an actor. The first type of behaviour function 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.

Upon close inspection, it can be seen that the `receive` function of the `StringCounter` actor in Figure 2.1, 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. Arriving messages are queued in its sole mailbox to be processed. Differently to the Erlang design, 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 to by other actors who are interested in particular event messages. Line 24 of the String Counter example demonstrates how to subscribe to messages in the event stream of an actor system.

2.4.1.4 Actor Creation with Props

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

Figure 2.3 gives part of the APIs of the `Props` class and its companion object. The `Props` class 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.

```
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 2.3: Akka API: Props

We have seen an example of creating a `Props` instance in Figure 2.1, that is:

```
  Props[StringCounter]
```

which is short for

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

The API of the first `Props.apply` method is carefully designed to take advantage of the Scala language. Firstly, the word `apply` can be omitted when 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:

```
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 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 2.1, 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 other 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 2.1, `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 2.4: 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 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 path*s 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 reference*s are equivalent if they correspond to the same actor path and the same actor. A 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 2.5 shows two groups of methods: those for interacting with other actors (lines 3 to 24), and those for controlling the behaviour of the

```
1  package akka.actor
2  trait ActorContext
3    abstract def actorOf(props: Props, name: String): ActorRef
4    abstract def actorOf(props: Props): ActorRef
5
6    abstract def child(name: String): Option[ActorRef]
7    abstract def children: Iterable[ActorRef]
8    abstract def parent: ActorRef
9
10   abstract def props: Props
11   abstract def self: ActorRef
12   abstract def sender: ActorRef
13
14   implicit abstract def system: ActorSystem
15
16   def actorFor(path: Iterable[String]): ActorRef
17   def actorFor(path: String): ActorRef
18   def actorFor(path: ActorPath): ActorRef
19   def actorSelection(path: String): ActorSelection
20
21   abstract def watch(subject: ActorRef): ActorRef
22   abstract def unwatch(subject: ActorRef): ActorRef
23
24   abstract def stop(actor: ActorRef): Unit
25
26   abstract def become(behavior: Receive,
27                       discardOld: Boolean = true): Unit
28   abstract def unbecome(): Unit
29   abstract def receiveTimeout: Duration
30   abstract def setReceiveTimeout(timeout: Duration): Unit
```

Figure 2.5: Akka API: Actor Context

represented actor (lines 26 to 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 in conflict with the name of another existing 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 the obtaining of actor references via a new method `actorSelection`, whose return 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 messages to a specific group of 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 manage 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 Dynamic Behaviour Swapping

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, the message handler of an actor is required to be updated at runtime.

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.

Figure 2.6 defines a calculator whose behaviour changes at run-time. The calculator starts with a basic version that can only compute multiplication.

```
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 2.6: Akka Behaviour Swap Example

When it receives an `Upgrade` command, it upgrades to an advanced version that can compute both multiplication and division. The advanced calculator downgrades 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.

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 actor creations. Recall that every user-created actor is initialised in one of 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.

```
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 2.7: Akka API: Supervisor Strategies

Figure 2.7 gives the APIs for Akka supervisor strategies. As in OTP, for each supervisor strategy, users can specify the maximum number of restarts permitted for its children within a period. The default supervisor strategy in Akka is `OneForOne` that permits unlimited restarts.

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

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

2.4.3 Case Study: A Fault-Tolerant Calculator

Figure 2.8 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

```
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 }
}
object SupervisedCalculator extends App {
  val system = ActorSystem("MySystem")
  val actorRef:ActorRef =
       system.actorOf(Props[SafeCalculator], "safecalculator")
  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 2)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 3)
}
/* Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
10 / 5 = 2
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
3 * 2 = 6
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
*/
```

Figure 2.8: Akka Example: Supervised Calculator

calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` is raised. The supervisor strategy of the safe calculator also specifies the maximum failures its child may have within a time range. If the child fails more frequently than the allowed frequency, the safe calculator will be stopped, and its failure will be reported to its supervisor, the system guardian actor in this example. The terminal output shows that the simple calculator is restarted before the third and fifth message are delivered. The last message is not processed since 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 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 implementation of the TAkka library.

2.5.1 Parameterized Types

A *parameterized type* $\texttt{T[U}_1\texttt{,...,U}_n\texttt{]}$ consists of a type constructor `T` and a positive number of type parameters $\texttt{U}_1\texttt{,...,U}_n$ [Odersky., 2013]. The type constructor `T` must be a valid type name whereas a type parameter $\texttt{U}_i$ 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 2.9 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*

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

```scala
class ArrayStack[E] extends Stack[E]{
  private var list:List[E] = Nil
  def empty():Boolean = {
    return list.size == 0
  }
  def push(elt:E):Unit = {
    list = elt :: list
  }
  def pop():E = {
    val elt:E = list.head
    list = list.tail
    return elt
  }
  override def toString():String = {
    return "stack"+list.toString.drop(4)
  }
}
```

```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(i)
  assert(stack.toString().equals("stack(3, 2, 1, 0)"))
  val top = stack.pop
  assert(top == 3 && stack.toString().equals("stack(2, 1, 0)"))
  val reverse = Stacks.reverse(stack)
  assert(stack.empty)
  assert(reverse.toString().equals("stack(0, 1, 2)"))
}
```

Figure 2.9: Scala Example: A Generic Stack Library

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`.

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 a `List` so that both prepending an element and removing the first element take constant time.

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 advantage of the Scala compiler called *autoboxing*, which converts a primitive type `Int` to its corresponding object wrapper class `Integer`. The example code uses 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 an expected type. In a sense, the `push(elt:E):Unit` method of `Stack[E]` specified in Figure 2.9 is overly restrictive because it only accepts an argument of type `E`, but not data of a subtype of `E`.

Figure 2.10 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 2.10, 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 2.10 is the same as the code in Figure 2.9

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

```scala
class ArrayStack[E] extends Stack[E]{
  private var list:List[E] = Nil
  def empty():Boolean = {
    return list.size == 0
  }
  def push[T <: E](elt:T):Unit = {
    list = elt :: list
  }
  def pop():E = {
    val elt:E = list.head
    list = list.tail
    return elt
  }
  override def toString():String = {
    return "stack"+list.toString.drop(4)
  }
}
```

```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(3, 2, 1, 0)"))
  val top = stack.pop
  assert(top == 3 && stack.toString().equals("stack(2, 1, 0)"))
  val reverse = Stacks.reverse(stack)
  assert(stack.empty)
  assert(reverse.toString().equals("stack(0, 1, 2)"))
}
```

Figure 2.10: Scala Example: A Generic Stack Library using Type Bounds

except that, on line 4 of the `Client` example, the value of `i` need to be explicitly converted to an `Integer`.

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 $\mathtt{Stack[T}_{sub}\mathtt{]}$ the subtype of `Stack[T]`, or the reverse? 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 using 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, and *get*s a return value from its output channel. According to the 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 2.11. 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 2.12. 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 parameterized type but is not available in scope. `OptManifest[+T]` is the supertype of `ClassManifest[T]` and `OptManifest`.

```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 2.11: Scala Example: A Covariant Immutable Stack

```
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 2.12: Scala API: Manifest Type Hierarchy

The code example in Figure 2.13 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 Function and Partial Function

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 2.14 gives a Scala example that illustrates 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, a completeness check of message patterns might be a useful feature in practice.

```
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 2.13: Scala ExampleI: Manifest Example

```
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 2.14: Scala Example: PartialFunction and Function

2.7 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 1998 when the Erlang/OTP library was 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 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 sophisticated 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 at the earliest opportunity to expose type errors.

Chapter 3

TAkka: Design and Implementation

(This chapter is expanded from [TAKKA], Chapter 4])

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 advantage 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 *statically* 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 which implements the claimed design. We outline how static and dynamic type checking are used to prevent ill-typed messages. Examples of TAkka applications show that type-parameterized actors can form supervision trees in the same way as actors without type parameters. This chapter concludes with a brief discussion about design alternatives used by other actor libraries.

The latest TAkka library is built on top of Akka 2.1.4. During the research of this project, TAkka has been built on stable Akka releases since 2.0. For all Akka versions we have tried, actors can be parameterized as expected. Nevertheless, as Akka APIs and the structure of Akka configuration file changes slightly in different Akka versions, readers who want to use a later Akka version may need to update the APIs or the configuration file according to the specification of the used Akka version.

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

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

The type of `m`, which is `String`, is omitted too because it can be inferred as well.

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

Thirdly, dynamic type checking is involved at the earliest opportunity when static type checking meets its limitation. For example, when a user looks up an actor reference by its type and path, as at line 21 and 23, the current TAkka design does not statically 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, an exception raises as soon as the ill-typed actor reference is claimed at the run time, earlier than the time when it is in use.

Because sending an actor 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 benefits 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 a distributed environment. For service developers, they can focus on the logic of the services rather than worrying about incoming messages of unexpected types.

```
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")
// dynamic type error!
  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

3.2 Type-parameterized Actor

A TAkka actor has type `TypedActor[M]`. It inherits the Akka `Actor` trait to minimize the implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor APIs. Instead, programmers are encouraged to use the typed interface 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 have 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]

  abstract 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 the Akka `Actor` class is `PartialFunction[Any, Unit]`. As mentioned in Section 2.6, an advantage of using `Function` is that the compiler can check the completeness of patterns when the input type of the function is a sealed trait or a sealed class. In Akka, pattern completeness checking is not considered because an Akka actor may receive messages of any type. In contrast, a TAkka actor only expects messages of a certain type. Therefore, pattern completeness checking is a helpful feature for TAkka users.

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 we are trying to avoid when possible, are still available to TAkka users. Unfortunately, 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. Re-implementing the TAkka Actor library requires a similar amount of work as 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 a 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 rationale. 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 is aiming at preventing 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 3.3. The `ActorRef` class takes two parameters: one type parameter that indicates the type of expected message and one implicit argument that records the `Manifest` of the type parameter. In most cases, the implicit `Manifest` can be provided by the Scala language automatically.

Like in Erlang and Akka, users send a message to an actor via the `!` method of its actor reference. 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 notation 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 3.4. The calculator defined in the example can compute the result of two types of operations: multiplication and division. Hence, `Division` 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 division requests only to that actor reference.

```scala
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 3.3: TAkka API: Actor Reference

For ease of use, `ActorRef` provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method encapsulates the process of type cast on `ActorRef`, a contravariant type. We believe that using the notation of the `publishAs` method can be more intuitive than thinking about contravariance and subtyping relationship each time, especially when publishing an actor reference as different types in a complex application. In addition, type conversion using `publishAs` is statically type checked. More importantly, with the `publishAs` method, users can give a supertype of an actor reference on demand, without defining new

```
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 3.4: TAkka Example: Actor Reference is Contravariant

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.

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 3.5: 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 code, Scala 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` instance 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 3.6, 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.

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 an IP address and a 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 a `NotRemoteSystemException`. Calling `remoteActorOf` takes longer time than calling `actorOf` because the IP address and the port number need to be fetched from the system configuration. The way to enabling distributed programming will be explained in Section 3.11.

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 residing in. TAkka removes APIs that enquire on the actor reference of an actor's parent and children for two reasons. Firstly, the types of the parent and children of an actor are unknown to the library developer as they vary 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 the `sender` because its type changes for each message. We recommend the Erlang-style message pattern, in which the actor reference to the message sender is part of the message if the sender expects a reply message.

The two `actorFor` methods are used for fetching an actor reference of the expected type located at an actor path. The task of type checking and reference fetching is delegated to the actor system (Section 3.8), which implements the same APIs.

```
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]

  @throws(classOf[NotRemoteSystemException])
  def remoteActorOf[Msg](props:Props[Msg])
                    (implicit mt:Manifest[Msg]):ActorRef[Msg]
  @throws(classOf[NotRemoteSystemException])
  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 watch, unwatch, and stop

def become[SupM >:  M](newTypedReceive:  SupM =>
    Unit,newSystemMessageHandler:SystemMessage =>
    Unit,newpossiblyHarmfulHandler:akka.actor.PossiblyHarmful =>
    Unit)(implicit smt:Manifest[SupM]):ActorRef[SupM]

  // no unbecome

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

Figure 3.6: TAkka API: Actor Context

The method signature of the `setReceiveTimeout` method and the `receiveTimeout` method are the same as the Akka version. The `setReceiveTimeout` method sets a timeout within which a new message is expected to be received. The `receiveTimeout` method returns the set timeout. In Akka, if no message is received within the specified timeout, a `ReceiveTimeout` *message* is sent to the actor itself. In Akka, the `ReceiveTimeout` message and other messages are handled by the `receive` method. In TAkka, the `ReceiveTimeout` message is handled by the `systemMessageHandler` method separately. Section 3.10 explains the TAkka design in depth.

Finally, TAkka defines the `become` method with a new signature so that hot swapping on the message handler is backward compatible. To guarantee backward compatible hot swapping, the `unbecome` method is removed. The next section explains hot swapping in TAkka.

3.6 Backward Compatible Hot Swapping

Hot swapping describes the technique to replace system components without shutting down the system or causing significant interruption to the system. Hot swapping is a desired feature of distributed systems, whose components are typically developed and deployed separately. Unfortunately, hot swapping is not supported by the JVM, the platform on which the TAkka library runs. To support hot swapping on an actor's receive function and the system message handler, those two behaviour methods are maintained as object references.

The `become` method enables hot swapping on the behaviour of an actor. The `become` method in TAkka is different from behaviour upgrades in Akka in two aspects. Firstly, as the handler for system messages are separated from the handler for other messages, TAkka users may update the system message handler as well. Secondly, hot swapping in TAkka *must* be backward compatible and *cannot* be rolled back. In other words, an actor must evolve to a version that is able to handle the original message patterns. The above decision is made so that a service published to users will not be unavailable later.

The `become` method is implemented as shown in Figure 3.7. The design of the `become` method involves both static type checking and dynamic type checking. The static type parameter `M` should be interpreted as the least general type of messages expected by an actor of type `TypedActor[M]`, whose initial message handler has a function type `M=>Unit`. When a user decides to upgrade the message handler of an actor, it is important to make sure that the new

```
package takka.actor

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

  def become[SupM >: M](
      behavior: SupM => Unit
  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] ={
    become(behavior, this.systemMessageHandler,
        this.possiblyHarmfulHandler)
  }

  def become[SupM >: M](
      behavior: SupM => Unit,
      newSystemMessageHandler:SystemMessage=>Unit
  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] = {
    become(behavior, newSystemMessageHandler,
        this.possiblyHarmfulHandler)
  }

  def become[SupM >: M](
      newTypedReceive: SupM => Unit,
      newSystemMessageHandler:
              SystemMessage => Unit
      newpossiblyHarmfulHandler:akka.actor.PossiblyHarmful => Unit
  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] = {
     val smt = manifest[SupM]
     if (!(smt >:> mt))
       throw BehaviorUpdateException(smt, mt)

    this.mt = smt
    this.systemMessageHandler = newSystemMessageHandler
    this.possiblyHarmfulHandler = newpossiblyHarmfulHandler

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

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

Figure 3.7: Hot Swapping in TAkka

message handler is aware of all types of messages that may be sent to actor before the upgrade. Backward compatible hot swapping requires that the input type of a new message handler should be a supertype of the input type of the old message handler. Unfortunately, the concrete type of the new message handler will only be known when the `become` method is invoked. When a series of `become` invocations are made at the run time, the order of those invocations may be non-deterministic. Therefore, it is not possible to guarantee backward compatibility by using static type checking only. As a result, dynamic type checking is required. To do so, each `TypedContext` instance record the mainifest of the input type of the current message handler. The recorded manifest is used to check if the input type of the new message handler is a *supertype* of the input type of the current message handler. If so, both the manifest and the message handler are updated. Although static type checking meets its limitation, it prevents some invalid `become` invocations at compile time.

The code in Figure 3.8 demonstrates code swapping on message handler in TAkka. The code is similar to the Akka example in Figure 2.6. The calculator server begins with a basic version. The basic version can only compute multiplication but leaves the developer an opportunity to upgrade it later. The `CalculatorUpgrade` test simulates a simple scenario. After using the basic calculator server for a while, the service developer implements an advanced message handler, `advancedCalculator`, which supports both multiplication and division. The developer updates the server by sending an `Upgrade` message that contains the new message handler. When the upgrading has been completed, users can send `Division` messages to the server.

There are two differences between the TAkka version (Figure 3.8) and the Akka version (2.6). First, two new types, `Operation` and `BasicOperation` are introduced. The `BasicOperation` trait is defined to be used as the type parameter of the `CalculatorServer` class. The `Operation` trait is not required for the basic calculator. The `Operation` trait is provided so that the developer could define new types of operation in addition to those basic ones. Second, there is no `Downgrade` message in the TAkka version. A user who has an actor reference of type `ActorRef[Operation]` always ensure that a division request can be understood by the server.

```scala
package sample.takka
import takka.actor.{ActorRef, ActorSystem, Props, TypedActor}

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

  case class Division(m:Int, n:Int) extends Operation
  def advancedCalculator:Operation=>Unit = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
    case Upgrade(_) =>
      println("Upgraded.")
  }
  simpleCal ! Upgrade(advancedCalculator)
  val advancedCal =
      system.actorFor[Operation]("akka://CalculatorSystem/user/calculator")
  advancedCal ! Multiplication(5, 3)
  advancedCal ! Division(10, 3)
  advancedCal ! Upgrade(advancedCalculator)
}
/* Terminal Output:
5 * 1 = 5
Upgrading ...
5 * 3 = 15
10 / 3 = 3
Upgraded.
 */
```

Figure 3.8: TAkka Behaviour Upgrade Example

3.7 Typed Name Server

In distributed systems, a name server maps each registered name, usually a unique string, to a dynamically typed value, and provides a function to look up a value for a given name. A name can be encoded as a `Symbol` in Scala so that names which represent the same string have the same value. As a value retrieved from a name server is *dynamically typed*, it needs to be checked against and be cast to an 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. Our typed name server can be straightforwardly ported to other platforms that support type reflection.

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 supertype of the most precise type of that value. In Scala, `TSymbol` and `TValue` can be simply defined as in Figure 3.9.

In the Scala implementation, `TSymbol` is declared as a *case class* 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. The `hashCode` method of `TSymbol` does not consider the value of its type descriptor for efficiency considerations.

3.7.2 Operations

With the notion of `TSymbol`, a typed name server provides the following three operations:

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

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

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

```scala
package takka.nameserver

import scala.collection.mutable.HashMap
import scala.reflect.Manifest
import scala.Symbol

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]
}

object NameServer {
  private val nameMap = new HashMap[TSymbol[_], TValue[_]]

  def set[T:Manifest](name:TSymbol[T], value:T):Boolean = synchronized {
    val tValue = TValue[T](value)
    if (nameMap.contains(name)){ return false }
    else{
      nameMap.+=((name, tValue))
      return true
    }
  }

  def unset[T](name:TSymbol[T]):Boolean = synchronized {
    if (nameMap.contains(name) && nameMap(name).t <:< name.t ){
      nameMap -= name
      return true
    }else{ return false }
  }

  def get[T](name:TSymbol[T]):Option[T] = synchronized {
    if (!nameMap.contains(name)) {return None}
    else {
      val tValue = nameMap(name)
      if (tValue.t <:< name.t) { return
          Some(tValue.value.asInstanceOf[T]) }
      else{ return None }
    }
  }
}
```

Figure 3.9: Typed Name Server

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

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

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

The TAkka library implements the typed name server using the code given in Figure 3.9. The typed name server internally saves and fetches data by using a standard hashmap data structure. The APIs are designed with the following considerations.

Firstly, when an operation fails, the name server returns `false` or `None` rather than raising an exception. The decision is made so that the name server is always available. Keeping a name server alive is important especially if the name server permits distributed enquiries, in which case the remote caller would like to have feedback. Although it seems that the current name server is only available to applications running in the same JVM, as we will see in Section 3.8, distributed enquiries can be easily supported via another application that supports distributed communication, for example, by using a TAkka actor.

Secondly, the `unset` method and the `get` method succeed as long as the associated type of the input name is a supertype of the associated type of the registered name. In other words, a user must know how the value is registered. For the `get` method, the returned value shall be used as a supertype of the registered type, which may have less methods. To permit polymorphism while keeping the efficiency of using hashmap, the `hashCode` method of `TSymbol` does not take its type manifest into account. Equivalence comparison on `TSymbol` instances, however, considers the type. The `hashCode` method of `TSymbol` ignores its type description so that it has an additional benefit. As typed symbols that have the same symbolic representation have the same hash value, we prevent the case where users accidentally register two typed names with the same symbolic representation but have different types, in which case if one type is a supertype of the other, the return value of `get` may be non-deterministic. In our design, only names that have not been registered can be added to the name server. Therefore, the `set` method does not need to check the type information as in the `unset` method and the `get` method. Because the type information in `TSymbol` is ignored in the hashmap, it is recorded in the notion of `TValue`, which does not appear in the APIs for library users.

Lastly, the implementations of the three operations are thread safe. They are `synchronized` to prevent thread interference and memory consistency errors. When one thread is executing one of the methods of `NameServer`, all other threads that invoke its methods are suspended. The `nameMap` is a private field to `NameServer` so that other objects cannot directly access it. Finally, the three simple operations are executed without interactions with other objects. Therefore, there is no liveness issue of the implementation.

3.7.3 Dynamic Type Comparison

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 uses the second method because it detects type errors which may otherwise be left out in the first method. Our implementation requires the runtime type reification feature provided by Scala. In a system that does not support type reification, implementing typed name servers is more difficult.

3.8 Look up Typed Actor References via Actor System

Same as in Akka, a TAkka actor system organises related actors in a tree structure and provides services such as thread scheduling, network connection, and logging. The APIs of the TAkka `ActorSystem` class is given in Figure 3.10. Because the functionality of a TAkka actor system is almost the same as an Akka `ActorSystem`, the TAkka `ActorSystem` is implemented by managing an Akka `ActorSystem` as its private field, to which dynamically typed tasks are delegated. In addition to using the Akka library functions, a TAkka `ActorSystem` employs both static and dynamic type checking to detect type errors. This section explains how the typed name server, described in the last section, is used when create and fetch type parameterized actor references.

In addition to delegating some tasks to an Akka actor system, a TAkka actor system uses the typed name server in the same JVM to save typed actor references created by that actor system. Each TAkka actor system initializes an

ActorTypeChecker instance that is responsible to enquire on whether a typed actor reference is registered at the typed name server located at the JVM it runs. In this section, Figure 3.11 presents the code of the actorOf function which creates an actor and registers its actor reference to the typed name server. Figure 3.12 presents the code of the actorFor function which fetches typed actor references.

```scala
package takka.actor

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

object ActorSystem
    def apply():ActorSystem
    def apply(sysname: String): ActorSystem
    def apply(sysname: String, config: Config): ActorSystem
    def apply(sysname: String, config: Config,
              classLoader: ClassLoader): ActorSystem

abstract class ActorSystem
    private 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],
                                 name:String):ActorRef[Msg]

    @throws(classOf[NotRemoteSystemException])
    def remoteActorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
    @throws(classOf[NotRemoteSystemException])
    def remoteActorOf[Msg:Manifest](props:Props[Msg],
                                       name:String):ActorRef[Msg]

    def actorFor[M:Manifest](actorPath: String): ActorRef[M]
    def actorFor[M:Manifest](actorPath: akka.actor.ActorPath):
        ActorRef[M]
```

Figure 3.10: TAkka API: Actor System

The actorOf method and the remoteActorOf method create an actor that is supervised by a user actor created by the system. The task of creating an actor is delegated to an Akka actor system, which is a private field of a TAkka actor system. The additional work done by the TAkka actor system is to register

```
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 {
  private val system:akka.actor.ActorSystem
  @throws(classOf[NotRemoteSystemException])
  def host:String
  def actorOf[Msg:Manifest](props:Props[Msg],
                            name:String):ActorRef[Msg] = {
    val actor = new ActorRef[Msg] {
      val untypedRef = system.actorOf(props.props, name) }
    NameServer.set(TSymbol[ActorRef[Msg]](Symbol(actor.path.toString())),
                   actor)
    actor
  }
  @throws(classOf[NotRemoteSystemException])
  def remoteActorOf[Msg:Manifest](props:Props[Msg],
                                  name:String):ActorRef[Msg] = {
    val akkaactor = actorOf[Msg](props:Props[Msg], aname:String)
    val actor = new ActorRef[Msg] {
      val localPathStr = akkaactor.path.toString()
      val takka_system = this
      val sys_path = localPathStr.split("@")
      val remotePathStr =
        sys_path(0)+"@"+takka_system.host+":"+takka_system.port+sys_path(1)
      //e.g. akka://RemoteCreation@129.215.91.195:2554/user/...
      val untypedRef = system.system.actorFor(remotePathStr)
    }
    NameServer.set(TSymbol[ActorRef[Msg]](scala.Symbol(actor.path.toString())),
                   actor)
    actor
  }

  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
          case Some(_) => sender ! Compatible
  } }}
}
```

Figure 3.11: Actor System: actorOf

```scala
def actorFor[M:Manifest](actorPath: akka.actor.ActorPath):
    ActorRef[M]= {
    val isRemotePath = actorPath.address.host match {
      case None => false
      case Some(_) => true
    }

    if (isRemotePath) {
      val untyped_ref:akka.actor.ActorRef = system.actorFor(actorPath)
      //e.g.
          "akka://CalculatorApplication@127.0.0.1:2552/user/ActorTypeServer"
      val remoteChecker:akka.actor.ActorRef =
      system.actorFor("akka://"+actorPath.address.system+"@"
                                 +actorPath.address.host.get+":"
                                 +actorPath.address.port.get+"/user/ActorTypeServer")
      implicit val timeout = new akka.util.Timeout(10000) // 10 seconds
      val checkResult = remoteChecker ? Check(actorPath, manifest[M])
      var result:ActorRef[M] = null
      checkResult onSuccess {
        case Compatible =>
          result = new ActorRef[M]{
            val untypedRef = untyped_ref
          }
        case NonCompatible =>
          throw new Exception("ActorRef["+actorPath+"] does not exist
                  or does not have type ActorRef["+manifest[M]+"]")
      }
      result
    }else{
      // local actor reference, fetch from local name server
      NameServer.get(TSymbol[ActorRef[M]](scala.Symbol(actorPath.toString)))
      match {
        case None =>
          throw new Exception("ActorRef["+actorPath+"] does not exist
                   or does not have type ActorRef["+manifest[M]+"]")
        case Some(ref) =>
          new ActorRef[M]{
            val untypedRef = system.actorFor(actorPath)
          }
      }
    }
  }
```

Figure 3.12: Actor System: actorFor

the typed actor path and the typed actor reference into the typed name server running in the same JVM (line 17 and line 33 in Figure 3.11). Because an actor reference returned by the Akka `actorOf` method cannot be used remotely as it does not contain an IP address and a port number, the TAkka library defines a `remoteActorOf` method which returns a typed actor reference that contains information required for using it remotely. If remote communication is not enabled by the actor system, a `NotRemoteSystemException` will arise.

The `actorFor` method returns a typed actor reference if there is such one that has an expected typed and is located at a given path. When looking for a typed actor reference, the actor system first checks if the input actor path contains an IP address and a port number. If so, it sends a request to the `ActorTypeServer` actor in the remote actor system; otherwise it sends a request to the `ActorTypeServer` actor in the local machine. When an `ActorTypeServer` actor receives a request that asks if there is an actor reference of the expected type at a given path, it checks registered names at the typed name server in its JVM.

Although a typed name server defined in the current TAkka implementation can only be directly called by applications running on the same JVM. As the implementation of `actorFor` shows, it can indirectly receive remote requests via applications that support distributed communication, for example, TAkka actors. The design of a consistent global shared typed name server is left as a future extension.

3.9 Supervisor Strategies

The TAkka library uses the Akka supervisor strategies explained in Section 2.4.2: `OneForOne` and `AllForOne`. If a supervisor adopts the `OneForOne` strategy, it restarts its child when it fails. The failure of an actor will not affect its siblings. 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. The `RestForOne` supervisor strategy can be simulated by grouping related children and defines special messages to trigger actor terminations. The TAkka library does not implement the `RestForOne` strategy because it is not needed for the applications we consider.

Figure 3.13 gives APIs of supervisor strategies in TAkka. Actually, it is the same APIs as the Akka version given in Figure 2.7. TAkka reuses the Akka APIs because none of the supervisor strategies requires type-parameters as TAkka separates the message handler for system messages and the message handler for other messages.

```
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 3.13: TAkka API: Supervisor Strategies

A TAkka Safe Calculator example is given in Figure 3.14 as a reminiscence of the Akka Safe Calculator in Figure 2.8. Both examples define 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. A fault tolerant calculator, safe calculator, is defined as the supervisor of the simple calculator. The safe calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` 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 next message is delivered.

The TAkka implementation is modified from the Akka version with changes marked in blue. First, an `Operation` trait is introduced as the supertype of the `Multiplication` message and the `Division` message. Second, actor classes are parameterized by the type of messages they expected. Third, the `calculator` actor reference in TAkka can publish itself as an actor reference, `multiplicator`, which only accepts multiplication requests only. The supervisor strategy used in the TAkka implementation is exactly the same as the one in the Akka implementation.

```
package sample.takka
import takka.actor.{ActorRef, ActorSystem, Props, 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))
  }
}
class SafeCalculator extends TypedActor[Operation] {
  import language.postfixOps
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 2, withinTimeRange = 1 minute) {
      case _: ArithmeticException =>
        println("ArithmeticException Raised to: "+self)
        Restart
    }
  val child:ActorRef[Operation] =
         typedContext.actorOf(Props[Operation, Calculator], "child")
  def typedReceive = {  case m => child ! m }
}
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
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
}
/*  Terminal Output:
3 * 2 = 6
3 * 3 = 9
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
10 / 5 = 2
*/
```

Figure 3.14: TAkka Example: Safe Calculator

3.10 Handling System Messages

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

Message	TAkka 2.0	TAkka 2.1	Akka 2.0	Akka 2.1
Kill	Y	Y	Y	Y
PoisonPill	Y	Y	Y	Y
ReceiveTimeout	Y	Y	Y	Y
ChildTerminated	Y	N	Y	N
Restart	Y	N	Y	N
Terminated	N	N	Y	Y
Create	N	N	Y	N
Failed	N	N	Y	N
Link	N	N	Y	N
Unlink	N	N	Y	N
Suspend	N	N	Y	N
Resume	N	N	Y	N

Table 3.1: System Messages

Table 3.1 lists system messages used in different versions of Akka and TAkka. The purpose of those messages are examples as the followings.

- `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.

- `ReceiveTimeout`

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

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

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

- `Restart`

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

- `Terminated`

  When an actor monitors the life cycle of another actor using Akka Death Watch [Typesafe Inc. (b), 2012, Section 3.1], the watcher will receive a Terminated(watched) message when the *watched* actor is terminated.

- `Create`

  A message sent to the created actor itself.

- `Failed`

  An actor send itself a `Failed(cause: Throwable)` message when an error of `cause` occurred when it is processing messages.

- `Link`

  A message sent to linked actors when Akka Death Watch [Typesafe Inc. (b), 2012, Section 3.1] is used.

- `Unlink` A message sent to linked actors when Akka Death Watch [Typesafe Inc. (b), 2012, Section 3.1] is disabled.

- `Suspend`

  A message sent by the system to an actor asking it suspend the process of processing remaining messages in its mailbox.

- `Resume` A message sent by the system to an actor to dissolve the effects of the `Suspend` message so that the actor will resume the process of processing messages in its mailbox.

The `ReceiveTimeout` message is retained because it is used in many applications considered in this thesis. The TAkka library retains `Kill` and `PoisonKill` because they are used when implementing the Chaos Monkey library and the Supervision View library explained in Section 5.5. The TAkka library does not retain `Create`. `Suspend`, and `Resume` because they are not used in applications considered in this thesis. Those messages can be included in future versions of TAkka if required. The TAkka library does not retain `Terminated`, `Link`, and `Unlink` because, as will be explained in Section 3.12, life-cycle monitor relationship outside the supervision tree is considered as a redundant design.

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

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

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

3.11 Case Study: A Distributed Calculator

In previous sections, we have seen that an actor can be parameterized by the type of messages it expects. Adding type parameters to actors does not affect the construction of supervision trees because system messages are separated from other messages. Because the TAkka library delegates the tasks of actor creation and message sending to the underlying Akka system, distributed communication can be done in TAkka in the same way as in Akka.

The example used in this section is modified from the example in [Typesafe Inc. (b), 2012, Section 5.11]. In this example, two calculators will be created as actors, one basic calculator that can compute addition and subtraction, one advanced calculator that can compute multiplication and division. The basic calculator will be created locally as previous examples. The advanced calculator will be created at a remote machine by updating the actor system configuration.

Actor references for local and remote actors are retrieved in the same way.

3.11.1 Actor System Configuration for Distribution

Application developers can override the default configuration of an Akka actor system by providing an alternative `Config` object or load the configuration from an `application.conf` file in the application deployment folder. [Typesafe Inc. (b), 2012] The configuration of a TAkka actor system is modified by changing the `Config` object which is used by the underlying Akka actor system. The details of akka system configuration are explained in the Akka documentation. This section only explains the configuration used in the distributed calculator example. For details of Akka actor system configuration, readers are suggested to look at the Akka documentation for the version the reader uses.

The configuration in Figure 3.15 is used for the `RemoteCreation` system in Figure 3.18. The configuration overrides three system policies. Firstly, the system enables the distributed communication by replacing the actor reference provider from `LocalActorRefProvider` to `RemoteActorRefProvider`. Secondly, the `deployment` block specifies that actor created at the logical path `/advancedCalculator` shall be physically created by the `CalculatorApplication` actor system located at address `129.215.91.88:2552`. Finally, the actor system itself is located at address `129.215.91.195:2554`. In this example, `129.215.91.88` and `129.215.91.195` are two IP addresses allocated for the Ethernet connection at the author's office. `2552` and `2554` are port numbers that are not used by the two computers used for this test.

```
akka{
 actor {
   provider = "akka.remote.RemoteActorRefProvider"
   deployment {
      /advancedCalculator {
        remote = "akka://CalculatorApplication@129.215.91.88:2552"
   }}}
   remote {
      netty {
        hostname = "129.215.91.195"
        port = 2554
   }
 }
}
```

Figure 3.15: Configuration Example: Distributed Creation

3.11.2 A Complete Example

The classes defined for the example described at the beginning of this section are the followings:

Operations and Messages The Operations (`MathOp`) considered in this example are addition (`Add`), subtraction (`Subtract`), multiplication (`Multiply`), and division (`Divide`). There are two types of messages used in this example: the `CalculatorMessage` sent to a real calculator that can computes an operation; the `MathResult` sent to a broker who receives calculation requests and display the result of each calculation.

Calculators The two calculator actors defined in this example are the `SimpleCalculatorActor` in Figure 3.17 and the `AdvancedCalculatorActor` in Figure 3.16. The simple calculator can compute addition and subtraction while the advanced calculator can compute multiplication and division.

Test Applications There are three test applications in this example. The `CalApp` application creates an actor system with name `CalculatorApplication` located at address `129.215.91.88:2552`. Inside the actor system, a simple calculator is created. The `CreationApp` application creates two actors: one `CreationActor` at the local machine and one `AdvancedCalculatorActor` at a remote node. The `CreationActor` is used as a broker that sends a request to the advanced calculator and print the returned result. Finally, the `LookupApp` application works as the same as the `CreationApp` except that the remote actor used in this application is the simple calculator fetched from the `actorFor` method.

The example code shows that distributed programming is TAkka is enabled in the same way as in Akka, that is, by updating the actor system configuration. For an Akka application that enables distributed programming, the same actor system configuration can be reused in the corresponding TAkka version.

```scala
package typed.remote.calculator
import takka.actor.{TypedActor, ActorRef}
import akka.actor.ActorPath
import scala.reflect.runtime.universe._

trait MathOp
case class Add(nbr1: Int, nbr2: Int) extends MathOp
case class Subtract(nbr1: Int, nbr2: Int) extends MathOp
case class Multiply(nbr1: Int, nbr2: Int) extends MathOp
case class Divide(nbr1: Double, nbr2: Int) extends MathOp

trait CalculatorMessage
case class Op(op:MathOp, sender:ActorRef[MathResult])
     extends CalculatorMessage

trait MathResult
case class AddResult(nbr: Int, nbr2: Int, result: Int)
     extends MathResult
case class SubtractResult(nbr1: Int, nbr2: Int, result: Int)
     extends MathResult
case class MultiplicationResult(nbr1: Int, nbr2: Int, result: Int)
     extends MathResult
case class DivisionResult(nbr1: Double, nbr2: Int, result: Double)
     extends MathResult
case class Ask(calculator:ActorRef[CalculatorMessage], op:MathOp)
     extends MathResult



class AdvancedCalculatorActor extends TypedActor[CalculatorMessage] {
  def typedReceive = {
    case Op(Multiply(n1, n2), sender) =>
      println("Calculating %d * %d".format(n1, n2))
      sender ! MultiplicationResult(n1, n2, n1 * n2)
    case Op(Divide(n1, n2), sender) =>
      println("Calculating %.0f / %d".format(n1, n2))
      sender ! DivisionResult(n1, n2, n1 / n2)
  }
}
```

Figure 3.16: Distributed Calculator: Messages and Advanced Calculator

```scala
package typed.remote.calculator
import akka.kernel.Bootable
import takka.actor.{ Props, TypedActor, ActorSystem }
import com.typesafe.config.ConfigFactory

class SimpleCalculatorActor extends TypedActor[CalculatorMessage] {
  def typedReceive = {
    case Op(Add(n1, n2), sender) =>
      println("Calculating %d + %d".format(n1, n2))
      sender ! AddResult(n1, n2, n1 + n2)
    case Op(Subtract(n1, n2), sender) =>
      println("Calculating %d - %d".format(n1, n2))
      sender ! SubtractResult(n1, n2, n1 - n2)
  }
}
class CalculatorApplication extends Bootable {
  val system = ActorSystem("CalculatorApplication",
               ConfigFactory.parseString("""
                 akka{
                   actor {
                     provider = "akka.remote.RemoteActorRefProvider"
                   }
                   remote {
                     netty {
                       hostname = "129.215.91.88"
                       port = 2552
                     }
                   }
                 }""") )
  val cal = system.actorOf(Props[CalculatorMessage,
                              SimpleCalculatorActor], "simpleCalculator")
}

object CalcApp {
  def main(args: Array[String]) {
    new CalculatorApplication
    println("Started Calculator Application - waiting for messages")
  }
}
```

Figure 3.17: Distributed Calculator: Simple Calculator Local Creation

```scala
package typed.remote.calculator
import akka.kernel.Bootable
import com.typesafe.config.ConfigFactory
import scala.util.Random
import takka.actor._

class CreationApplication extends Bootable {
  val system = ActorSystem("RemoteCreation",
                        ConfigFactory.parseString("""
                akka{
                  actor {
                    provider = "akka.remote.RemoteActorRefProvider"
                    deployment {
                      /advancedCalculator {
                        remote =
                              "akka://CalculatorApplication@129.215.91.88:2552"
                  }}}
                  remote {
                    netty {
                      hostname = "129.215.91.195"
                      port = 2554
                }}}""") )
  val localActor = system.actorOf(Props[MathResult, CreationActor],
                                                    "creationActor")
  val remoteActor = system.actorOf(Props[CalculatorMessage,
                        AdvancedCalculatorActor], "advancedCalculator")

  def doSomething(op: MathOp) = { localActor ! Ask(remoteActor, op) }
}
class CreationActor extends TypedActor[MathResult] {
  def typedReceive = {
    case Ask(calculator, op) => {
      calculator ! Op(op, typedRemoteSelf)
    }
    case result: MathResult => result match {
      case MultiplicationResult(n1, n2, r) =>
        println("Mul result: %d * %d = %d".format(n1, n2, r))
      case DivisionResult(n1, n2, r) =>
        println("Div result: %.0f / %d = %.2f".format(n1, n2, r))
}}}

object CreationApp extends App {
    val app = new CreationApplication
    while (true) {
      if (Random.nextInt(100) % 2 == 0)
         app.doSomething(Multiply(Random.nextInt(20),
                                 Random.nextInt(20)))
      else app.doSomething(Divide(Random.nextInt(10000),
                                 (Random.nextInt(99) + 1)))
      Thread.sleep(200)
}}
```

Figure 3.18: Distributed Calculator: Distributed Creation

```scala
package typed.remote.calculator
import akka.kernel.Bootable
import scala.util.Random
import com.typesafe.config.ConfigFactory
import takka.actor.{ ActorRef, Props, TypedActor, ActorSystem }

class LookupApplication extends Bootable {
  val system = ActorSystem("LookupApplication",
                          ConfigFactory.parseString("""
                akka{
                  actor {
                    provider = "akka.remote.RemoteActorRefProvider"
                  }

                  remote {
                    netty {
                      hostname = "129.215.91.195"
                      port = 2553
                    }
                  }
                }""") )
  val actor = system.actorOf(Props[MathResult, LookupActor],
      "lookupActor")
  val remoteActor = system.actorFor[CalculatorMessage]
      ("akka://CalculatorApplication@129.215.91.88:2552/user/simpleCalculator")

  def doSomething(op: MathOp) = { actor ! Ask(remoteActor, op) }
}
class LookupActor extends TypedActor[MathResult] {
  def typedReceive = {
    case Ask(calculator, op) => { calculator ! Op(op, typedRemoteSelf) }
    case result: MathResult => result match {
      case AddResult(n1, n2, r)  =>
          println("Add result: %d + %d = %d".format(n1, n2, r))
      case SubtractResult(n1, n2, r) =>
          println("Sub result: %d - %d = %d".format(n1, n2, r))
    }
  }
}
object LookupApp extends App {
    val app = new LookupApplication
    while (true) {
      if (Random.nextInt(100) % 2 == 0)
         app.doSomething(Add(Random.nextInt(100), Random.nextInt(100)))
      else
          app.doSomething(Subtract(Random.nextInt(100),
                                        Random.nextInt(100)))
      Thread.sleep(200)
    }
}
```

Figure 3.19: Distributed Calculator: Actor Look up

3.12 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 3.20 shows 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 the type pollution problem, explained in Section 5.1, by using Akka typed actors is as awkward as using a plain object-oriented model, where supertypes need to be defined in advance. In Scala and Java, introducing a supertype in a type hierarchy requires modification to all affected classes, whose source code may not be accessible by application developers.

Actors with or without Internal Mutable States The actor model formalized by Hewitt et al. [1973] does not specify its implementation strategy. In Erlang, a functional programming language, actors do not have mutable states. The state of an actor, if there is any, is recommended to be saved in an *ETS* table, a data structure provided by the OTP library. [Ericsson AB., 2013b] 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 author of this thesis encourages 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 [Fielding and Taylor, 2002], are more likely to achieve a good scalability and availability.

```
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 3.20: Akka Example: String Counter using Akka TypedActor

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.

3.13 Summing Up

This chapter presents the design and implementation of the TAkka library. In TAkka, an actor reference is parameterized by the type of the messages expected by the actor. Similarly type parameter is added to the `Actor` class, the `Prop` class and the `ActorContext` class. The TAkka library uses both static and dynamic type checking so that type errors are detected at the earliest opportunity. To enable looking up on remote actor references, TAkka defines a typed name server that keeps maps from typed symbols to values of the corresponding types.

The TAkka library adds type checking features to the Akka library but delegates tasks such as actor creation and message passing to the underlying Akka systems. This chapter shows that, by separating the handler for system messages and the handler for other messages, supervision tree and remote communication can be done in the same way as in the Akka library.

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 2.1) 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 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 4.1, 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 2.1. That is, no changes are required for the client application.

TAkka actor reference delegates the task of message sending to an Akka actor reference, its `untypedRef` field. In line 31 in Figure 2.1, 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.

4.2 Akka Actor in TAkka system

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

Users can initialize a TAkka actor reference by providing an Akka actor reference and a type parameter. In Figure 4.2, we re-use the Akka actor, initialize the actor in an Akka actor system, and obtained an Akka actor reference as in Figure 2.1. Then, we initialize a TAkka actor reference, `takkaServer`, which only accepts `String` messages.

```
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

This section evaluates the TAkka library regarding to the following three aspects. First, Section 5.1 shows that the notion of type-parameterized actor in the TAkka library has syntactical advantages to avoid the Wadler's type pollution problem. Second, Section 5.2 and 5.4 show that rewriting Akka programs using TAkka will *not* bringing obvious runtime and code-size overheads. Third, Section 5.5 gives two accessory libraries, ChaosMonkey and SupervisionView, for testing the reliability and availability of TAkka applications.

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 that are constructed using the layered architecture [Dijkstra, 1968; Buschmann et al., 2007] or the MVC pattern [Reenskaug, 1979, 2003] 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].

The other solution is publishing a component as of different types to different parties. The published type shall be a supertype of the most precise type of the component. In Java and Scala applications, this solution could be tricky because, as briefly discussed in Section 5.1.3, a set of supertypes must be defined in advance. Fortunately, if the component is implemented as a type-parameterized actor, the limitation can be avoided straightforwardly.

The demonstration example studied in this section is a Tic-Tac-Toe game with graphical user interface implemented using the MVC pattern.

5.1.1 Case Study: Tic-Tac-Toe

5.1.1.1 The Game

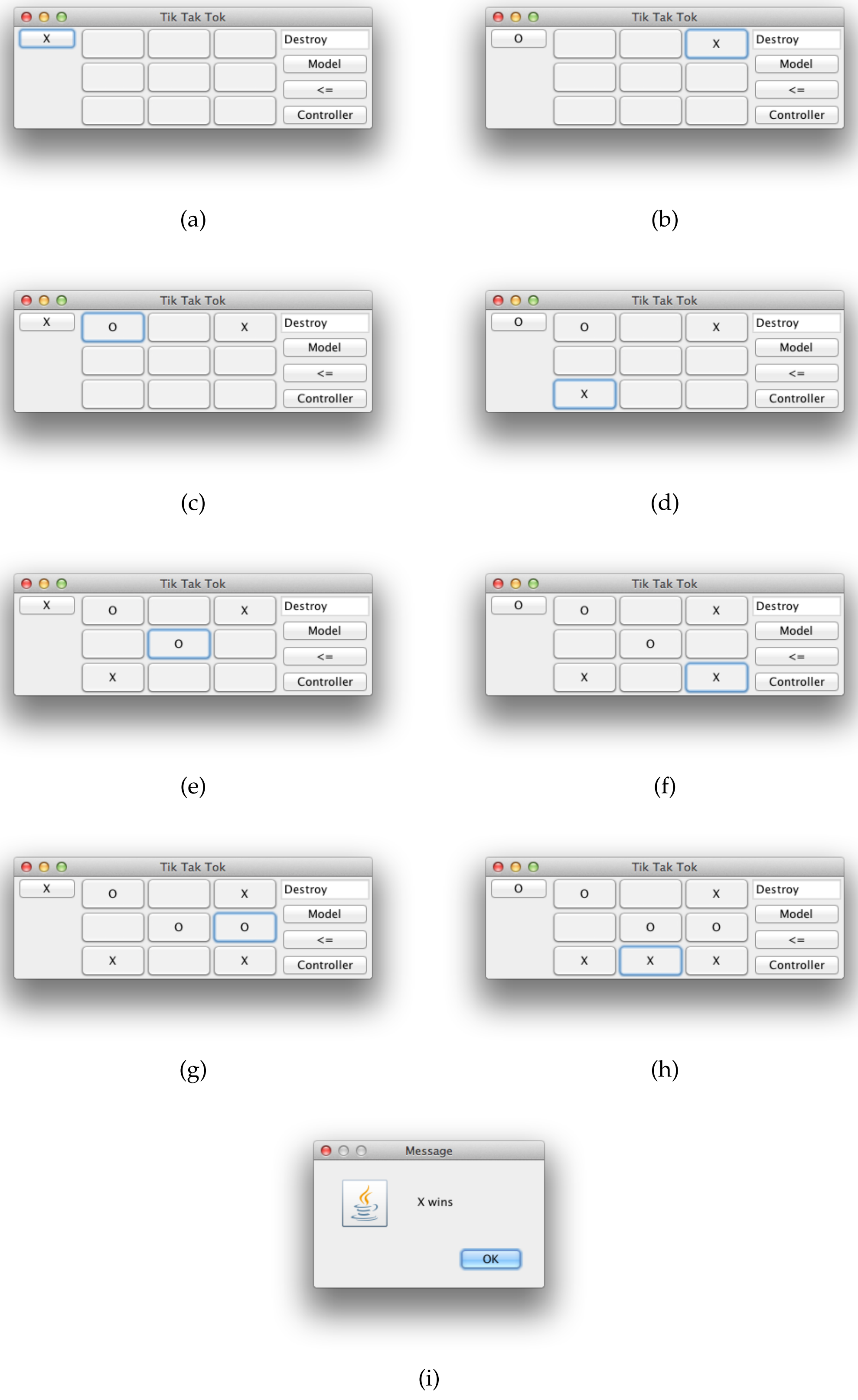
Tic-Tac-Toe [Wikipedia, 2013d], also known as Noughts and Crosses, is a paper-and-pencil game. A basic version of the Tic-Tac-Toe game is played by two players who mark X and O in turn in a 3×3 grid. A player wins the game if he succeeds in placing three respective marks, i.e. three Xs or three Os, in a horizontal, vertical, or diagonal row. The game is draw if no player wins when the grid is fully marked.

Figure 5.1 gives an example game won by the first player, X. Figure 5.1a to 5.1h are screenshots of the game implemented in the next subsection. The graphical user interface of the game contains three parts. The left hand side of the window shows the player of the next move. The middle of the window shows the current status of the game board. The right hand side contains control buttons. Finally, Figure 5.1i announces the winner.

5.1.1.2 The MVC pattern

Model-view-controller (MVC) is a software architecture pattern introduced by Reenskaug [1979]. The pattern separates the data abstraction (the model), the representation of date (the view), and the component for manipulating the date and interpreting user inputs (the controller). One key design principle of MVC is that the model and the view *never* communicate to each other directly. Instead, the controller is responsible for coordinating the model and the view, for example, sending instructions and reacting to messages from the model and the view.

MVC has been widely used in the design of applications with a graphical user interface (GUI), from early Smalltalk programs written in Xerox Parc [Reenskaug, 1979, 2003], to modern web application frameworks like the Zend framework [Allen et al., 2009], to mobile applications including Apple iOS applications [Apple Inc., 2012]. In this section, we will follow the MVC pattern to implement a Tic-Tac-Toe Game as a Scala application with a graphical user interface. The challenge here is using types to prevent the case when a model sent the controller a message expected from a view, and the case that a view pretends to be a model.



X wins

Figure 5.1: A Game of Tic-Tac-Toe

5.1.2 A TAkka Implementation

TAkka solves the type pollution problem by using polymorphism. The code from Figure 5.2 to Figure 5.5 gives an TAkka application that implements the Tic-Tac-Toe game with a GUI. The code marked in the blue color may be reused by other applications built using the MVC pattern.

Messages used in this implementation are given in Figure 5.2. Messages sent to the controller are separated into two groups: those expected from the model and those expected from the view. The `Controller` actor of this application, defined in Figure 5.5, reacts to messages sent either from the `Model` actor or the `View` actor. In its initialization process, however, the controller publishes itself as different types to the view actor and the model actor. Although the `publishAs` methods in line 22 and line 23 of Figure 5.5 can be committed because the type of the controller has been refined in the `ModelSetController` message and the `ViewSetController` message, we explicitly express the type convention and let the compiler double check the type.

In the definition of the `Model` actor (Figure 5.3) and the `View` actor (Figure 5.4), the `Controller` actor is declared as different types. As a result, both the view and the model only know the communication interface between the controller and itself. The `Model` actor internally represents the game board as a two dimensions array. Each time the model receives a request from the controller, it updates the status of the board and then announce the winner or the next player to the controller. The `View` actor maintains a GUI application that displays the game board and listens to user input. All user input are forwarded to the controller which sends corresponding requests to the model. When the view receives requests from the controller, it updates the game board or announce the winner via the GUI.

Finally, setting up the application is straightforward. In the code given at the bottom of Figure 5.5, a `Model` actor, a `View` actor, and a `Controller` actor are initialized in a local actor system. In this implementation, the controller actor must be initialized at the end because its initialization requires actor references of the model and the view. The user interface of this application looks like the one gives in Figure 5.1.

```
package sample.tic_tac_toe.takka

sealed trait ControllerMessage
sealed trait View2ControllerMessage extends ControllerMessage
final case class ButtonClickedAt(row:Int, col:Int) extends
    View2ControllerMessage

sealed trait Model2ControllerMessage extends ControllerMessage
final case class GridNotEmpty(row:Int, col:Int) extends
    Model2ControllerMessage
final case class PlayedCross(row:Int, col:Int) extends
    Model2ControllerMessage
final case class PlayedO(row:Int, col:Int) extends
    Model2ControllerMessage
final case class NextMove(move:Move) extends Model2ControllerMessage
final case class Winner(move:Move) extends Model2ControllerMessage

sealed trait Controller2ViewMessage
final case class DisplyError(err:String) extends Controller2ViewMessage
final case class DrawCross(row:Int, col:Int) extends
    Controller2ViewMessage
final case class DrawO(row:Int, col:Int) extends Controller2ViewMessage
final case class DisplayNextMove(move:Move) extends
    Controller2ViewMessage
final case class AnnounceWinner(winner:Move) extends
    Controller2ViewMessage

sealed trait Controller2ModelMessage
final case class MoveAt(row:Int, col:Int) extends
    Controller2ModelMessage

final case class
    ModelSetController(controller:ActorRef[Model2ControllerMessage])
    extends Controller2ModelMessage
final case class
    ViewSetController(controller:ActorRef[View2ControllerMessage])
    extends Controller2ViewMessage


sealed trait Move
final case object X extends Move
final case object O extends Move
```

Figure 5.2: TicTacToe: Message

```scala
package sample.tic_tac_toe.takka
import takka.actor._
final class Model extends TypedActor[Controller2ModelMessage] {
 var controller:ActorRef[Model2ControllerMessage] = _
 def typedReceive = {
   case ModelSetController(control) => controller = control
   case MoveAt(row:Int, col:Int) => { model.setStatus(row, col) }
 }
 private object model {
   sealed trait GridStatus
   case object Empty extends GridStatus
   case object XModelMove extends GridStatus
   case object OModelMove extends GridStatus // Uppercase O

   var nextXMove:Boolean = true // true->X false->O
   val status:Array[Array[GridStatus]] =
         Array(Array(Empty, Empty, Empty),
               Array(Empty, Empty, Empty),
               Array(Empty, Empty, Empty))
   def setStatus(row:Int, col:Int) = {
     if(nextXMove){
         if (status(row)(col) == Empty) {
           status(row)(col) = XModelMove
           controller ! PlayedCross(row, col)
           nextXMove = false
           controller ! NextMove(O)
         }else{ controller ! GridNotEmpty(row, col)    }
     }else{
         if (status(row)(col) == Empty) {
           status(row)(col) = OModelMove
           controller ! PlayedO(row, col)
           nextXMove = true
           controller ! NextMove(X)
         }else{ controller ! GridNotEmpty(row, col)    }
     }

     checkWinner match {
       case Empty =>
       case XModelMove =>      controller ! Winner(X)
       case OModelMove =>      controller ! Winner(O)
    }}
  def checkWinner:GridStatus = {
    // reuse GridStatus instead of a new set of values
    // return XModelMove if X wins
    // return OModelMove if O wins
    // return Empty if no winner has
}}
```

Figure 5.3: TicTacToe: Model

```scala
package sample.tic_tac_toe.takka

import takka.actor._
import scala.swing._
import scala.swing.event._
import javax.swing.JOptionPane

final class View extends TypedActor[Controller2ViewMessage]{
  private var controller:ActorRef[View2ControllerMessage] = _

  private var guiApp:GUIApplication = _;

  def typedReceive = {
     case ViewSetController(control) =>
       assert(controller == null, "controller has been set")
       controller = control
       guiApp = new GUIApplication(controller)
       guiApp.main(Array(""))
     case DisplyError(err) =>   guiApp.displayError(err)
     case DrawCross(row, col) =>  guiApp.draw(row, col, true)
     case DrawO(row, col) =>   guiApp.draw(row, col, false)
     case DisplayNextMove(move) =>  guiApp.showNextMove(move)
     case AnnounceWinner(winner:Move) => winner match{
       case X => guiApp.announceWinner(true)
       case O => guiApp.announceWinner(false)
     }
  }
}


class GUIApplication(controller:ActorRef[View2ControllerMessage])
      extends SimpleSwingApplication {
   def draw(row:Int, col:Int, isCross:Boolean) {
     // draw X or O at (row, col)
   }
   def showNextMove(move:Move) {
     // update next player
   }
   def displayError(err:String){
     // show error message
   }
   def announceWinner(isCross:Boolean){
     // announce winner
   }
}
```

Figure 5.4: TicTacToe: View

```scala
package sample.tik_tak_tok.takka

import takka.actor._

final class Controller(model:ActorRef[Controller2ModelMessage],
    view:ActorRef[Controller2ViewMessage]) extends
    TypedActor[ControllerMessage] {
  def typedReceive = {
    case ButtonClickedAt(row, col) =>
      model ! MoveAt(row, col)
    case GridNotEmpty(row, col) =>
      view ! DisplyError("grid "+row+" , "+col+" is not empty")
    case PlayedCross(row, col) =>
      view ! DrawCross(row, col)
    case PlayedO(row, col) =>
      view ! DrawO(row:Int, col:Int)
    case NextMove(move) =>
      view ! DisplayNextMove(move)
    case Winner(move) =>
      view ! AnnounceWinner(move)
  }

  override def preStart() = {
    model !
        ModelSetController(typedSelf.publishAs[Model2ControllerMessage])
    view !
        ViewSetController(typedSelf.publishAs[View2ControllerMessage])
  }
}

package sample.tic_tac_toe.takka

import takka.actor._
object TicTacToeApplication extends App {
  val system = ActorSystem("LocalTicTacToe")
  val model = system.actorOf(Props[Controller2ModelMessage, Model],
      "model")
  val view = system.actorOf(Props[Controller2ViewMessage, View], "view")
  val controller = system.actorOf(Props(new Controller(model, view)),
      "controller")
}
```

Figure 5.5: TicTacToe: Application

5.1.3 A Scala Interface

The type pollution problem is avoided in TAkka by publishing different types of an actor to different users. This method can be applied to any language that supports polymorphism. For example, Figure 5.6 gives an Interface of implementing the Tic-Tac-Toe game using the MVC pattern. The code marked in the blue color can be modified for building other applications using the MVC pattern.

Similar to the TAkka implementation which separates the type of messages sent from a model to a controller and a view to a controller, the interface in Figure 5.6 separates the methods of a controller to be called by a model and those to be called by a view in two distinct traits. The controller is defined as the subclass of both traits.

The example implementation, however, is difficult to maintain. Notice that the `ControllerForView` trait and the `ControllerForModel` trait are the supertypes of the `Controller` trait. As a result, those two traits and their methods should be defined in advance. Each time a new method is added to any of two traits due to the changes of application requirement, the whole program needs to be recompiled ad re-deployed. In the case that an application is a collaborated projects maintained by different groups, attempts of updating such an application throughly shall be avoided when possible.

On the contrast, in our TAkka implementation, new messages can be added easily. With the benefit of backward compatible hot swapping on message message handlers, the controller, the model and the view can be updated separately.

```scala
package sample.tic_tac_toe.mvcobject

trait Controller extends ControllerForView with ControllerForModel
class GameController(model:Model, view:View) extends Controller {
  // implementation
}

trait ControllerForView {
  def buttonClickedAt(row:Int, col:Int):Unit
}
trait ControllerForModel {
  def gridNotEmpty(row:Int, col:Int):Unit
  def playedCross(row:Int, col:Int):Unit
  def playedO(row:Int, col:Int):Unit
  def nextMove(move:Move):Unit
  def winner(move:Move):Unit
}

trait Model {
  def setController(controller:ControllerForModel):  Unit
  def moveAt(row:Int, col:Int): Unit
}
class GameModel extends Model {
  // implementation
}

trait View {
  def setController(controller:ControllerForView):  Unit
  def displyError(err:String): Unit
  def drawCross(row:Int, col:Int): Unit
  def drawO(row:Int, col:Int): Unit
  def displayNextMove(move:Move): Unit
  def announceWinner(winner:Move): Unit
}
class GameView extends View {
  // implementation
}

sealed trait Move
final case object X extends Move
final case object O extends Move

object TicTacToeApplication extends App {
  val model = new GameModel;
  val view = new GameView;
  val controller = new GameController(model, view)
}
```

Figure 5.6: TicTacToe: MVC Interface

5.2 Expressiveness and Correctness

5.2.1 Examples

To assess expressiveness and correctness of the TAkka library. 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.

5.2.1.1 Examples from the Quviq Project

Quviq [Arts et al., 2006] is a QuickCheck tool for Erlang programs. It generates random test cases according to specifications for testing applications written in Erlang. Quivq is a commercial product. The author gratefully acknowledge Thomas Arts from Quivq.com and Francesco Cesarini from Erlang Solutions for providing the Erlang source code for the ATM simulator example and the Elevator Controller example, two examples used in their commercial training courses.

The descriptions below reflects the design of the Akka and TAkka version ported from the Erlang source code. This thesis will only describe the overall structure of those two examples. The Akka and TAkka code is available in the ♠**private repository** ♠ but is not available in the ♠**public repository** ♠. Readers who would like to have access to the Erlang source code shall contact Quivq.com and Erlang Solutions directly.

ATM simulator This example contains 5 actor classes. It simulates a bank ATM system consists of the following components:

- a database backend that keeps records of all users.
- a front-end for the ATM with graphical user interface
- a controller for the ATM

Figure 5.7, cited from [Cesarini, 2011], gives the Finite State Machine that models the behaviour of the front-end for the ATM.

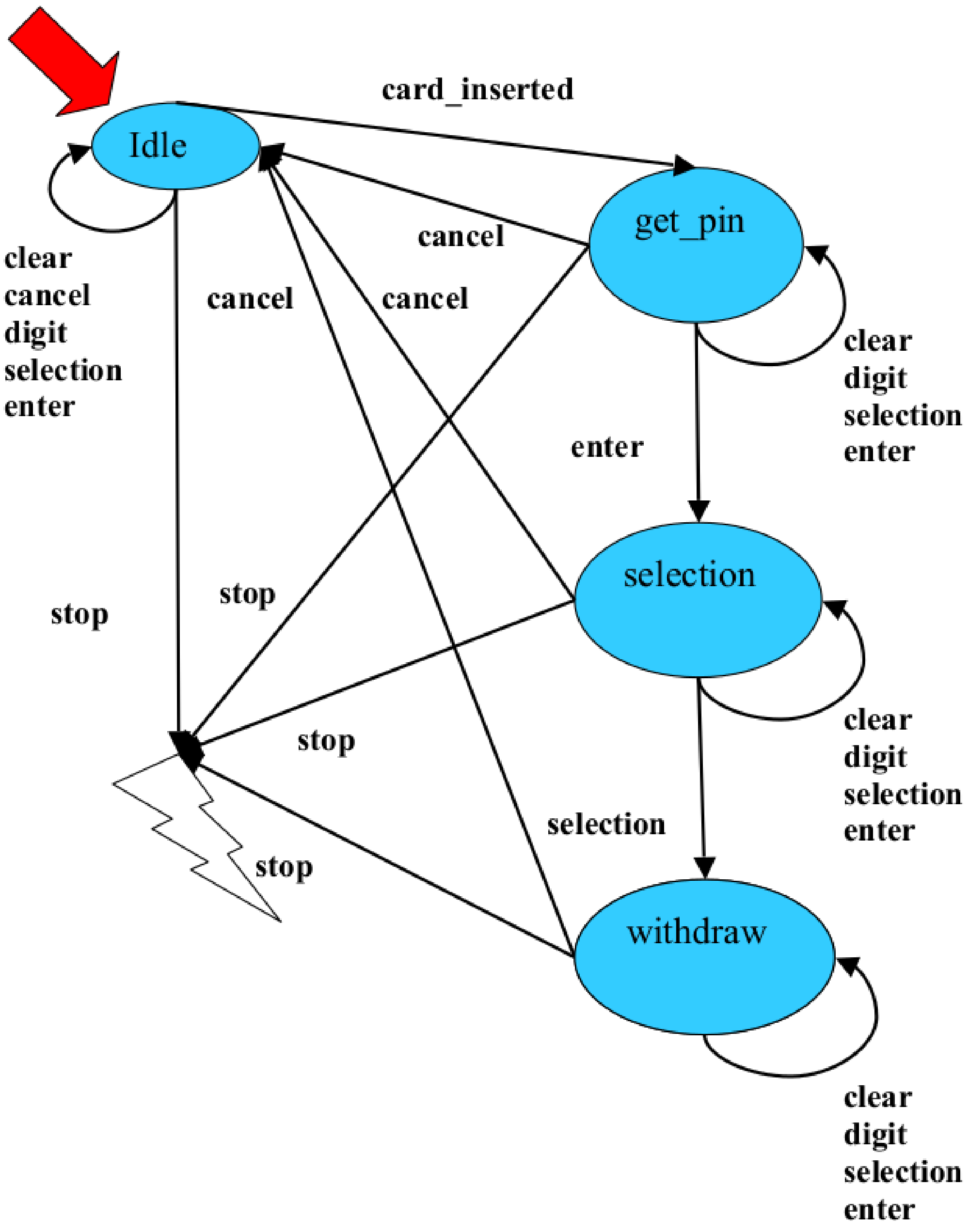


Figure 5.7: Example: ATM

Elevator Controller This example contains 7 actor classes. It simulates a system that monitors and schedules a number of elevators.

Figure 5.8 gives an example elevator controller that controls three elevators in a building that has 6 levels. The three worker actors are:

- The monitor class that provides a GUI.
- The elevator class that models a specific elevator.
- The scheduler class that reacts to user inputs.

The rest four actors are supervisors for other components. The example is shipped with QuickCheck properties that checks that events generated by users are correctly handled.

5.2.1.2 Examples from the Akka Documentation

The Akka DocumentatioTypesafe Inc. (b) [2012] contains some examples that demonstrate actor programming and supervision in Akka. We port the following examples to check that applications built using TAkka behaves similar to Akka applications.

Ping Pong This example contains two actor classes: `Ping` and `Pong`. Those two actors sending messages to each other for a number of times and then terminate. The example begins with a `ping` message sent from a `Ping` actor to a `Pong` actor. The `Pong` actor replies a `pong` message when it receives a `ping` message. When a `Ping` actor receives a `pong` message, it updates its internal message counter. If the message counter does not exceed a set number, it sends another `Ping` message, otherwise the program terminates.

Dining Philosophers This example contains two actor classes. It models the dining philosophers problem [Wikipedia, 2013a] using Finite State Machines (FSMs). The Dining Philosophers problem [Wikipedia, 2013a] is one of the classical problems that exhibits synchronization issues in concurrent programming. In the ported version five philosophers sit around at a table with five chopsticks interleaved. Each philosopher alternately thinks and eats. Before started eating, a philosopher needs to hold chopsticks on both sides. At any time, a chopstick can only be held by one philosopher. A philosopher put down both chopsticks when he finishes eating and think for a random period.

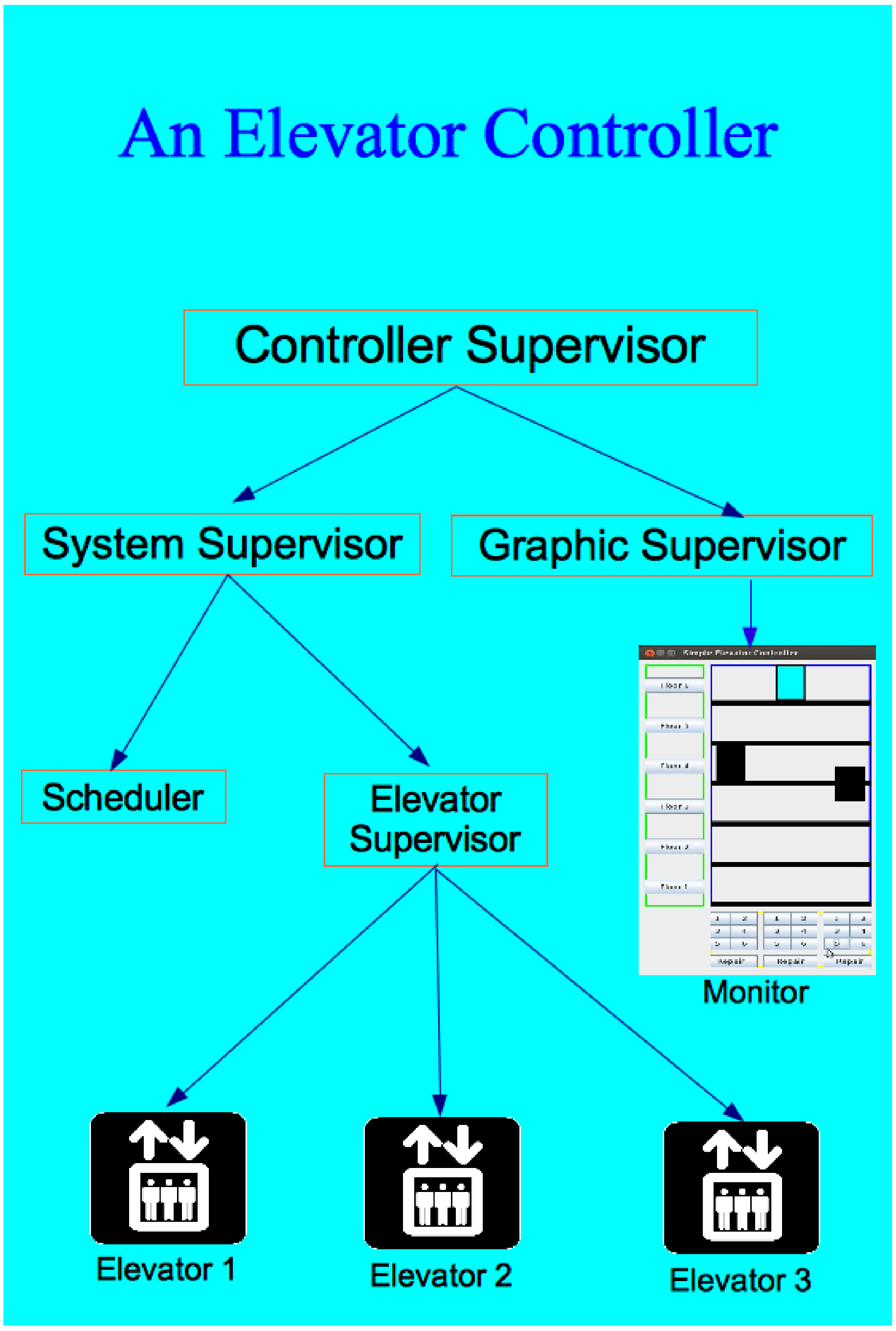


Figure 5.8: Example: Elevator Controller

Distributed Calculator This example contains four actor classes. It demonstrates distributed computations and hot code swapping on the receive function of an actor. The TAkka version of this example is used as a case study in Section 3.11.

Fault Tolerance This example contains 5 actor classes. It models a simple key-value data storage. The data storage maps `String` keys to `Long` values. The data storage throws a `StorageException` when users try to save a value between 11 and 14. The data storage service is supervised using the `OneForOne` supervisor strategy.

5.2.1.3 Examples from Other Open Source Projects

The Quiviq examples and the Akka documentation examples are demonstration examples for training purposes. We further port the following examples from open source projects to enlarge the scope of the test.

Barber Shop This application has six actor classes. The Akka version of this example is implemented by Zachrison [2012]. This example application models the Sleeping barber problem Wikipedia [2013c], which involves inter-process communication and synchronization.

EnMAS This medium size project has five actor classes. The EnMAS project, which stands for Environment for Multi-Agent Simulation, is a framework for multi-agent and team-based artificial intelligence research. [Doyle and Allen, 2012] Agents in this framework are actors while specifications are written in DSL defined in Scala.

Socko Web Server The implementation of this application contains four actor classes. Socko Imtarnasan and Bolton [2012] is a lightweight Scala web server that can serve static files and support RESTful APIs.

Gatling This application contains four actor classes. Gatling Excilys Group [2012] is a stress testing tool for web applications. It uses actors and synchronous I/O methods to improve the efficiency. The application is shipped with a tool that reports test results in graphical charts.

Play Core This large application only has one actor class in its core library. The Play framework Typesafe Inc. (c) [2013] is part of the TypeSafe stack for building web applications. The Play project is actively maintained by developers in TypeSafe Inc. and the Play community. Therefore, we only port its core library which is also updated less frequently.

5.2.2 Evaluation Results

5.2.2.1 Code Size

Following the suggestion by Fleming and Wallace [1986] and 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 5.1 show that when porting an Akka program to TAkka, about 7.4% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because TAkka code does not need to handle unexpected messages. On average, the total program size of Akka and TAkka applications are almost the same.

5.2.2.2 Type Error

A type error is reported by the compiler when porting the Socko example [Imtarnasan and Bolton, 2012] from its Akka implementation to an 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 pass 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.

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 5.1: Results of Correctness and Expressiveness Evaluation

5.3 Throughput

The Play example [Typesafe Inc. (c), 2013] and the Socko example [Imtarnasan and Bolton, 2012] used in Section 5.2 are two frameworks for building web services in Akka. A scalable implementation of a web service should be able to have a higher maximum throughput when more web servers are added. Throughput is measured by the number of correctly requests handled per unit time.

The JSON serialization example from the TechEmpower Web Framework benchmarks [TechEmpower, Inc., 2013] checks the maximum throughput achieved during a test. This example is used in this thesis to test how the maximum throughput changes when adding more web server applications implemented using Akka Play, TAkka Play, Akka Socko, and TAkka Scoko.

For a valid HTTP request sent to path `/json`, e.g. the one given in Figure 5.9a, the web service should return a JSON serialization of a new object that maps the key message to the value “Hello, World”. JSON, which stands for JavaScript Object Notation, is a language independent format for data exchanging between applications [JSON ORG, 2013]. Figure 5.9b gives an example of expected HTTP response. The body of the example response, line 7, is the expected JSON message.

```
GET /json HTTP/1.1
Host: server
User-Agent: Mozilla/5.0 (X11; Linux x86_64) Gecko/20130501 Firefox/30.0
AppleWebKit/600.00 Chrome/30.0.0000.0 Trident/10.0 Safari/600.00
Cookie: uid=12345678901234567890;
__utma=1.1234567890.1234567890.1234567890.1234567890.12; wd=2560x1600
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
```

(a) An Example of HTTP Request

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json; charset=UTF-8
3 Content-Length: 28
4 Server: Example
5 Date: Wed, 17 Apr 2013 12:00:00 GMT
6
7 {"message":"Hello, World!"}
```

(b) An Example of HTTP Response

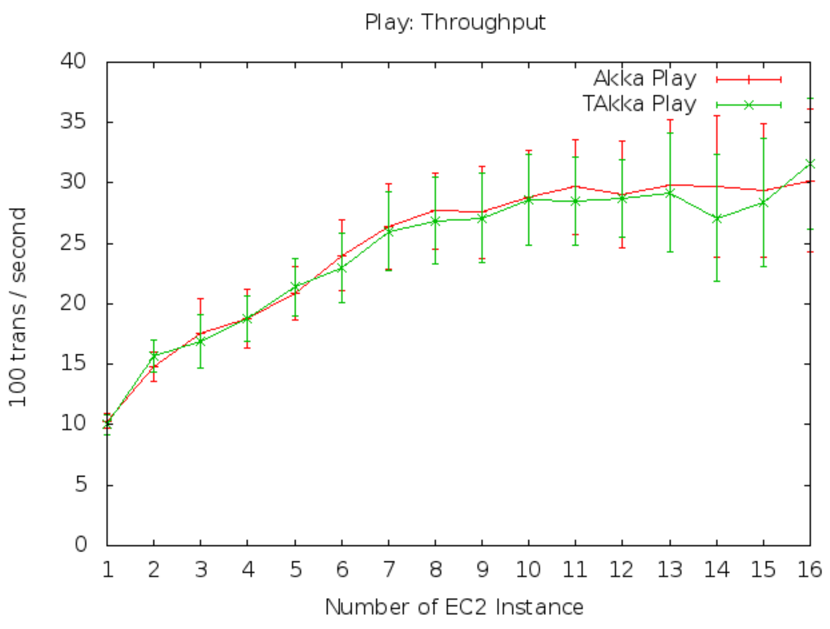
Figure 5.9: Example: JSON serialization Benchmark

All four versions of the web services are deployed to servers on Amazon Elastic Compute Cloud (EC2) [Amazon.com, Inc., 2013a]. The example is tested with up to 16 EC2 micro instances (t1.micro), each of which has 0.615 GB Memory. We expected that web servers built using Akka-based library and TAkka-based library have similar throughput.

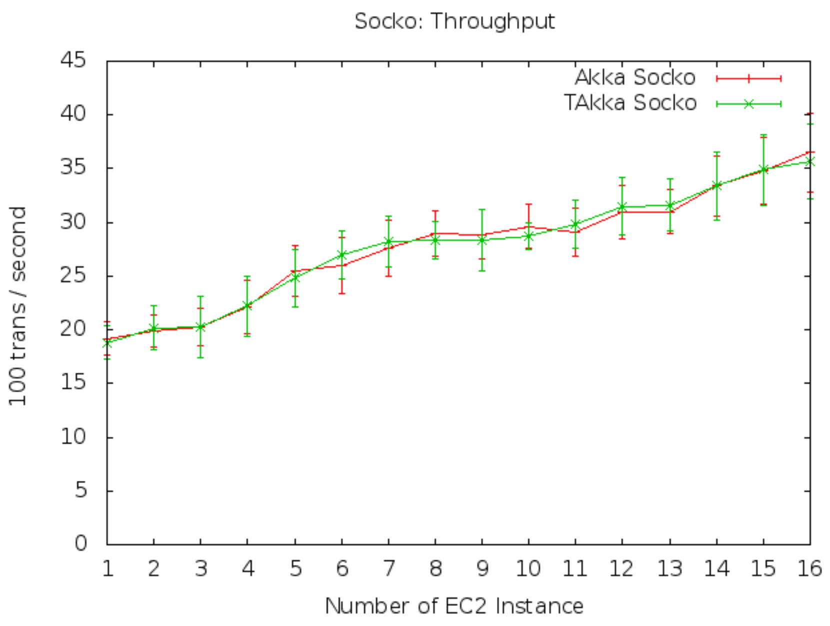
To avoid pitfalls mentioned in Amazon.com, Inc. [2012], we further design and implement the FreeBench [HE, 2013] tool, a benchmarking tool for HTTP servers. One feature of the FreeBench tool is that it can benchmark web servers deployed at multiple addresses. In the test, we confirmed that, for the JSON serialization example, the maximum throughput achieved when using the Elastic Load Balancing (ELB) service [Amazon.com, Inc., 2013b] is not significantly improved when more servers are added. On the other hand, when we benchmark all deployed EC2 servers, the total throughput achieved is increased slightly. One possible explanation for the above observation is that, the load balancer used in the first test, which runs on a micro EC2 instance, is the I/O bound of the throughput measurement. Another feature of the FreeBench tool is that it can be configured to carry out a number of benchmarks in parallel and repeat the parallel benchmark for a certain number of times. The benchmark results of all test are sent to a data store which reports a customised statistical summary.

The parameters set in this example are the number EC2 instances used. For each of the four types of servers, we test the example with up to 16 EC2 instances. For each number of EC2 instances, 10 rounds of benchmarking are executed. In each round, 20 sub-benchmarks are carried in parallel to maximise the utility of broadband. For each sub-benchmark, 10,000 requests are sent. We also manually monitors the upload and download speed during the test to confirm that the network speed is stable for most of time during the test with the above configurations.

Figure 5.10 summaries the result of the JSON serialization benchmark. It shows the average and the standard derivation of the throughput in each test. The result shows that web servers built using Akka-based library and TAkka-based library have similar throughput.



(a)



(b)

Figure 5.10: Throughput Benchmarks

5.4 Efficiency 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.

iv) the cost of dynamic type checking when registering new typed names and hot swapping message handlers.

We assess the upper bound of the cost of cost i) and ii) by a micro benchmark which assesses the time of initializing n instances of `StringCounter` defined in Figure 2.1 and Figure 3.1. When n ranges from 10^4 to 10^5, as shown in Figure 5.11, the TAkka implementation is about 2 times slower as the Akka implementation.

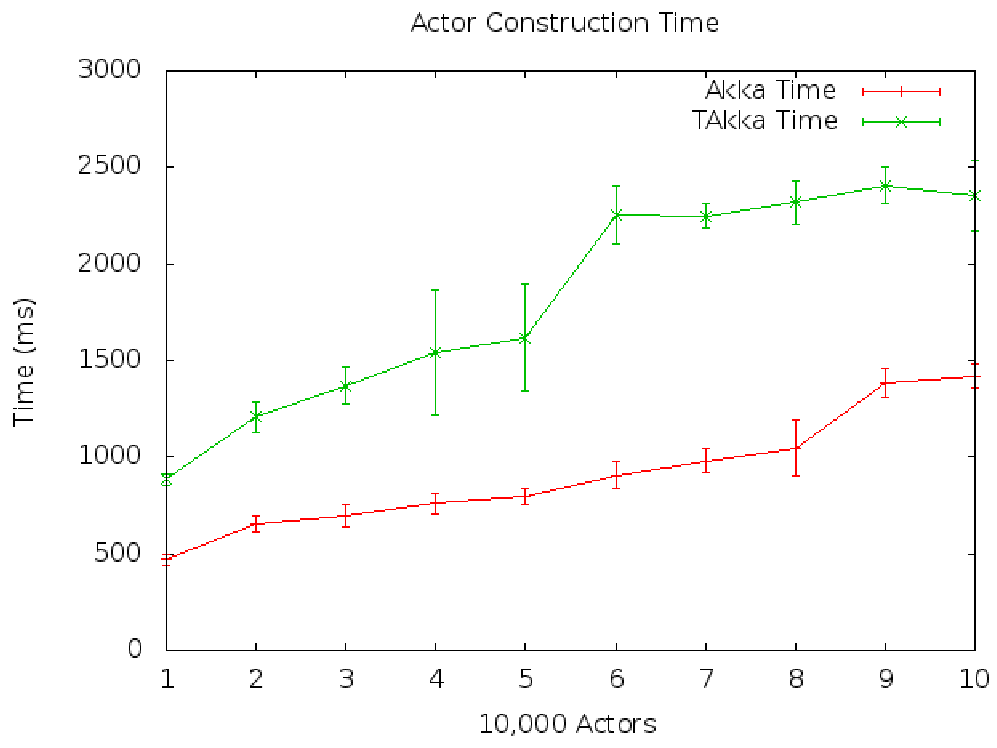


Figure 5.11: Cost of Actor Construction

The cost of transmitting a type descriptor should be close to the cost of transmitting the string representation of its fully qualified type name. The relative overhead the last cost depends on the cost of computations spent on application logic.

TAkka applications that have a relatively heavy computation cost shall have similar runtime efficiency and scalability compared with equivalent Akka applications because static type checking happens at compile time and dynamical type checking is usually not the main cost of applications that involve other meaningful computations. To confirm the above expectation, we further investigated the speed-up of multi-nodes TAkka applications by porting micro benchmark examples, listed from the BenchErl benchmarks in the RELEASE project Boudeville et al. [2012]; Aronis et al. [2012].

5.4.1 BenchErl Overview

BenchErl [Boudeville et al., 2012; Aronis et al., 2012] is a scalability benchmark suite for applications written in Erlang. It includes a set of micro benchmark applications that assess how an application changes its performance when additional resources (e.g. CPU cores, schedulers, etc.) are added. Thesis uses those BenchErl examples which does not use OTP ad-hoc libraries to investigate how the performance of an application changes when more distributed nodes are added.

All BenchErl examples are implemented in a similar structure. Each BenchErl benchmark spawns one master process and a configurable number of child processes. Child processes are evenly distributed across available potentially distributed nodes. The master process asks each child process to perform a task and send the result back to the master process. Finally, when results are collected from all child processes, the master process assembly those results and report the overall elapsed time of the benchmark.

BenchErl examples have similar structure to the MapReduce model proposed by Dean and Ghemawat [2008], which matches many real world tasks. More importantly, those programs are automatically parallelized when executed on a cluster of machines. This pattern allows benchmark users to focus on the effects of changes of computational resources rather than specific parallelization and scheduling strategies of each example.

5.4.2 Benchmark Examples

5.4.2.1 Ported BenchErl Examples

The following BenchErl examples are ported for comparing efficiency and scalability application built using TAkka and Akka:

bang This benchmark tests many-to-one message passing. The child processes spawned in this example are *sender* actors which sends the master process a fixed number of dummy messages. The master process initialize a counter set to the product of the number of processes and the number of messages expected from each child. When a dummy message is received, the master count down the number of remaining expected messages. The benchmark example completes when all expected messages are received.

Parameters set in this example are the number of available nodes, the number of child processes to spawn, and the number of messages sent by each child process. Instead of carrying out computations, the main task of this benchmark is sending messages from child processes to the master process. Therefore, the benchmark is likely to be I/O bounded when the number of child processes is large.

big This benchmark tests many-to-many message passing. A child process in this example sends a `Ping` message to each of the other child processes. Meanwhile, each child replies a `Pong` message when it receives a `Ping` message. Therefore, if n child processes are spawned, each child is expected to send $n-1$ messages and receive $n-1$ messages from others. When a child completed the task, it sends a `BigDone` message to the master actor. The benchmark example completes when the master actor receives `BigDone` messages from all of its children.

Parameters set in this example are the number of available nodes and the number of child processes to spawn. The main task of this benchmark is sending messages rather than computations. For each child process, the number of messages it sends and receives is linear to the total number of child processes. Same as the `bang` example, the benchmark is likely to be I/O bounded when the number of child processes is large.

ehb This benchmark re-implement the hackbench example [Zhang, 2008] originally for stress test for Linux schedulers. Each child process in this benchmark is a group of message senders and receivers. Each sender sends each receiver a dummy message and wait for an acknowledge message. Each sender repeats the process for a number of times. When a sender has received all expected replies, it reports to the child actor that it completes its task. When all senders in the group completes their tasks, the child process sends a complete message to the master process. The benchmark completes when all child processes finishes their tasks.

Parameters set in this example are the number of available nodes, the number of groups, group size, and the number of loops. Let n be the number of groups in this benchmark, and m be the number of senders and receivers in each group. The master process then expects n messages while a total of $2m^2$ messages are sent in each group. Therefore, the main task of this benchmark is sending messages inside each group. With the increased number of available nodes to share the task of child processes, this benchmark is expected to have shorter runtime.

genstress This benchmark is similar to the bang test. It spawns an echo server and a number of clients. Each client sends some dummy messages to the server and waits for its response. When a client receives the response, it sends an acknowledge message to the master process. The benchmark completes when results from all child processes are received. The Erlang has two versions, one using the OTP *gen_server* behaviour, the other implements a simple server-client structure manually. For generality, this benchmark ports the version without using *gen_server*.

Parameters set in this example are the number of available nodes, the number of child client processes to spawn, and the number of messages sent by each child process. The main task of this benchmark is sending messages from child processes to the master process. Therefore, the benchmark is likely to be I/O bounded when the number of child processes is large.

mbrot This benchmark models pixels in a 2-D image of a specific resolution. For each pixel at a given coordinate, the benchmark determines whether it belongs to the Mandelbrot set [Wikipedia, 2013b] or not. The determination process usually requires a large number of iterations. In this benchmark, child processes in this benchmark share roughly the same number of points. The benchmark completes when all child processes finishes their tasks.

Parameters set in this example are the number of available nodes, the number of child processes to spawn, and the dimensions of the image. Keeping the dimensions of the image to be a medium fixed size, with more available nodes to share the computation task, this benchmark is expected to have shorter runtime.

parallel This benchmark spawns a number of child processes. Each child process creates a list of N timestamps and checks that elements of the list are strictly increased, as promised by the implementation of the *now* function. After completing the task, the child process sends the result list to the master process. The benchmark completes when results from all child processes are received.

Parameters set in this example are the number of available nodes, the number of child processes, and the number of timestamps each child to create. Compared to the cost of creating timestamps and comparing data locally, the cost of sending distributed messages is usually much higher. Therefore, the runtime of this benchmark is likely to be bounded by the task of of sending results to the mater process.

ran This benchmark spawns a number of processes. Each child process generates a list of 100,000 random integers, sorts the list using quicksort, and sends the first half of the result list to the master process. The benchmark completes when results from all child processes are received.

Parameters set in this example are the number of available nodes and the number of child processes to spawn. For each child process, the cost of generating integers is linear to the number of integers, and the cost of sorting is linear logarithmic to the number of integers. If the number of generated integers in each child process is increased so that the cost of communicating with the master process can be neglected, this benchmark is a good example for scalability test. Unfortunately, the space cost of this example also increases when the number of generated integers is increased. In the TAkka and Akka benchmarks, the cost of garbage collection by JVM cannot be neglected when the number of generated integers is set to a higher number.

serialmsg This benchmark tests message forwarding through a dispatcher. This benchmark spawns one proxying process and a number of pairs of message generators and message receivers. Each message generator creates a random short string message and asks the proxying process to forward the message to a specific receiver. A receiver sends the master process a message when it receives a message. The benchmark completes when the master process receives messages from all receivers.

The parameters set in this example are the number available nodes, the number of pairs of senders and receivers, the number of messages and the message length. Clearly, this benchmark is I/O bounded when the speed of generating messages exceeds the speed of forwarding messages.

5.4.2.2 BenchErl Examples that are Not Ported

The following BenchErl examples are not ported for reasons given in respected paragraphs.

ets_test ETS table is an Erlang build-in module for concurrently saving and fetching shared global terms. [Ericsson AB., 2013b] This benchmark creates an ETS table. Child processes in this benchmark perform insert and lookup operations to the created ETS table for a number of times. This example is not ported because it uses ETS table, a feature that is specific to the Erlang OTP platform.

pcmark Same as the *ets_test* example, this benchmark also tests the ETS operations. In this benchmark, five ETS tables are created. Each created table is filled with some values before the benchmark begins. The benchmark spawns a certain number of child processes that read the content of those tables. This example is not ported as well because it uses ETS table.

timer_wheel Similar to the *big* example, this benchmark spawns a number of child processes that exchange ping and pong messages. Different from the *big* example, processes in this example can be configured to wait reply messages only for a specified timeout. In the case that no time out is set or is set to a short period, this example is the same as the *big* example. If a timeout is set to a short period, the runtime of this example is bounded by the timeout. For the above reason, this example is not ported.

5.4.3 Benchmark Methodology

5.4.3.1 Testing Environment

The benchmarks are run on a 32 node Beowulf cluster at the Heriot-Watt University. The 32 Beowulf cluster nodes each comprise eight Intel 5506 cores running at 2.13GHz. All machines running under Linux CentOS 5.5. The Beowulf nodes are connected with Baystack 5510-48T switch with 48 10/100/1000 ports.

5.4.3.2 Determining Parameters

The main interest of the efficiency and scalability test is to check whether applications built using Akka and TAkka have similar efficiency and scalability. Meanwhile, our secondary interest is to known how the required run-time of a BenchErl example changes when more number of machines are employed. Ideally, for each comparison on the efficiency of Akka application and equivalent TAkka application, the only variable should be the number of employed nodes. Nevertheless, those BenchErl examples listed in Section 5.4.2.1 has more parameters to be configured. Although experiments for our main interests can be carried out with any parameter, however; for the consideration of our secondary interest, parameters of each example are selected according to the following three criteria:

First, except the RUN example, the runtime of each experiment is controlled under 40 seconds. The decision is made so that the time to measure each example is acceptable. As we will explain in the next section, each example are tested for a total of 90 rounds of experiments. Another reason for this decision is that the experiment should be able to complete in a reasonable short time when running on a single machine. During the experiment, we find some configurations such that an example has a bad performance when runs on a single machine but speed up significantly when another machine shares part of its workload. Those experiments give interesting results that prove the importance of distributed programming, however; we would like the number of nodes be the only independent variable throughout all experiments. Therefore, we prefer the configuration that neglects the impact of other factors such as garbage collection.

Second, we prefer configurations that have more workload for each child process. In a number of tests to determine parameters, we observed that employing more number of machines only have runtime benefit for those

BenchErl examples whose runtime is bounded by the computational tasks rather than the I/O cost of communicating with the only master process. The only exception of this principle is the Parallel Fibonacci Numbers example for the verification of our observation.

Third, we prefer configurations that have more number of child processes but does not violate the above two principles. The benchmark is run on a maximum of 32 Beowulf machines. Although each machine has 8 CPU cores, the number of CPU cores used to execute Akka and TAkka applications is not guaranteed. For each example, we started with an small number of child processes. If the child process can have a higher workload by setting other parameters, we changes other parameters until the configuration violates the first criterion. If the number of child processes and the number of available nodes are the only two parameters, or the workload of each child process does not changes significantly with other possible configurations, we increase the number of child process gently until the example takes a long time to be completed in a single machine.

Base on the result of trial experiments, parameters used in each example are the followings:

bang

- number of child processes: 512
- number of messages sent by each child process: 2000

big

- number of child processes: 1024

ehb

- number of groups: 128
- group size: 10
- number of loops: 3

genstress

- number of child processes: 64
- number of messages sent by each child process: 300

mbrot

- number of child processes: 256
- dimensions of the image: 6000x6000

parallel

- number of child processes: 256
- number of timestamps each child to create: 6000

ran

- number of child processes: 6000
- list size: 100000

serialmsg

- number of pairs of senders and receiver: 256
- number of messages: 100
- message size: 200 characters

5.4.3.3 Measurement Methodology

After determining the benchmark parameters of each example, we measuring the runtime of each program as follows. First, each benchmark contains nine tests that uses different number of Beowulf nodes. The number of nodes used in the benchmark are 1, 4, 8, 12, 16, 20 , 24, 28, and 32. Similar to the Benchmark Harness process in [Blackburn et al., 2006], we run the test after a number of dry-run the benchmark for a couple of time to warm-up the runtime environment. After the warm-up period, the test is run for 10 times. The runtime is recorded for later analysis. Following guidance given by Fleming and Wallace [1986] and Hennessy and Patterson [2006], we report the efficiency of each example using a specific number of nodes by given the average and standard derivation of the 10 runs. The speed-up of a benchmark example using *n* nodes is measured as the proportion of the average time with one node and the average time with *n* nodes. After each test, we clean up the runtime environment before changing the number of nodes or switch to another benchmark example.

5.4.4 Evaluation Results

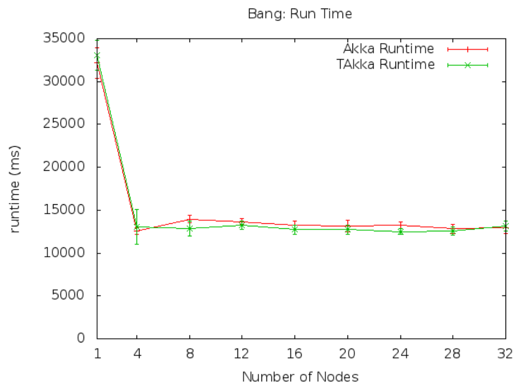
The records of the BenchErl benchmarks are summarised in Figure 5.12. For each benchmark example, we report its efficiency and speed-up when running on different number of nodes. The efficiency results include both the average runtime and the standard deviation. The scalability results are computed based on the average runtime. We observe the followings though the benchmarking.

Observation 1 In all examples, TAkka and Akka implementations have almost identical run-time and hence have similar scalability. In Figure 5.12, the runtime of Akka benchmarks and TAkka benchmarks often overlay on each other. For benchmarks that do not overlays, the difference is less than 10% on average. The scalability of Akka applications and the scalability of their TAkka equivalents appears slightly different because the differences of them is amplified by the differences of their runtime when running on a single node.

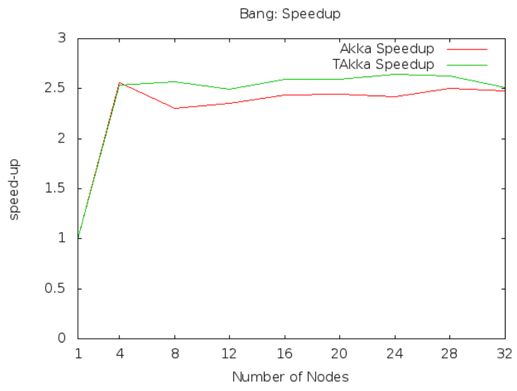
Observation 2 Some benchmarks scale well when more nodes are added. Examples of this observation are the `EHB` example and the `MBrot`.

Observation 3 Some benchmarks only scale well when a small number of nodes are added. Those example do not scale when the number of nodes are greater than a certain number. Examples of this observation are the `Bang` example, the `Big` example, and the `Run` example. The speed-up of those examples does not further increase when the number of nodes is more than four or eight.

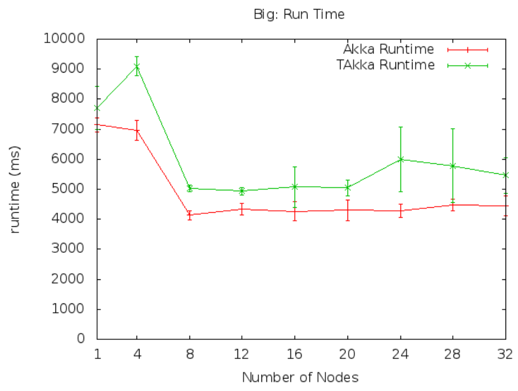
Observation 4 Some benchmarks do not scale. Examples of this observation are the `GenStress` example and the `Parallel` example.example, and the `SerialMsg` example.



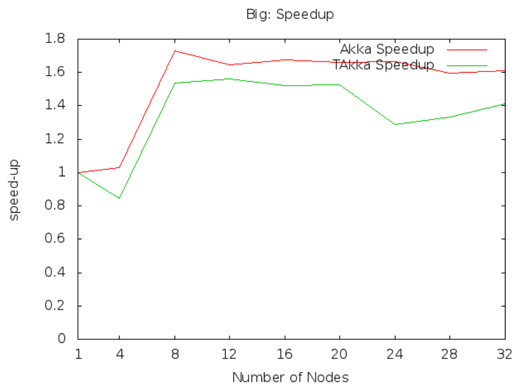
(a) Bang Time



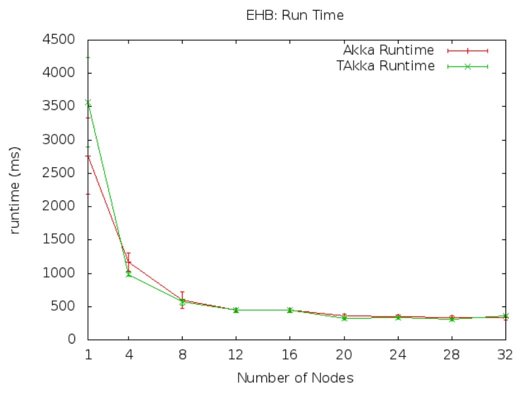
(b) Bang Scalability



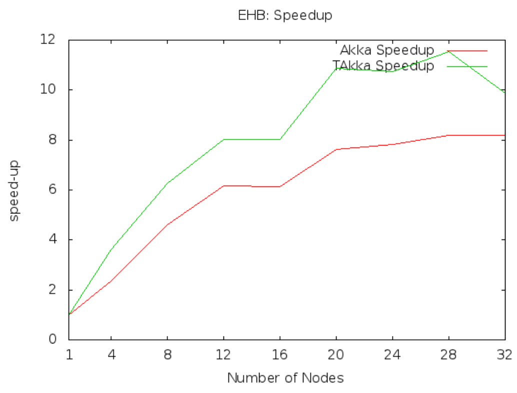
(c) Big Time



(d) Big Scalability

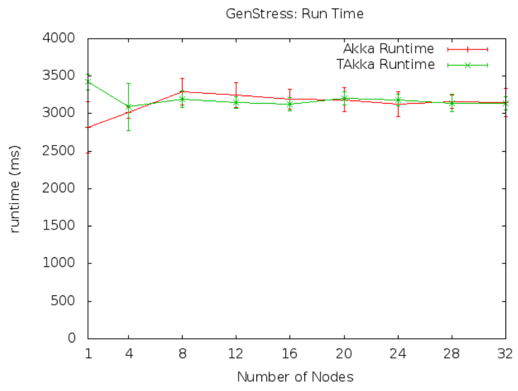


(e) EHB Time

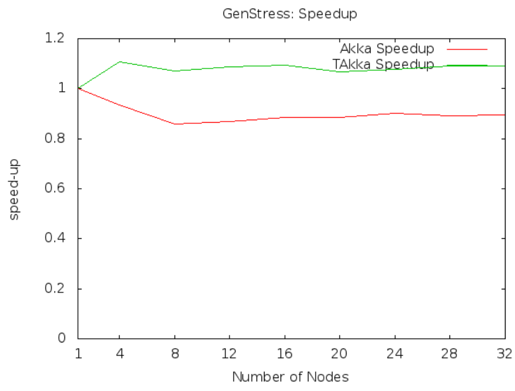


(f) EHB Scalability

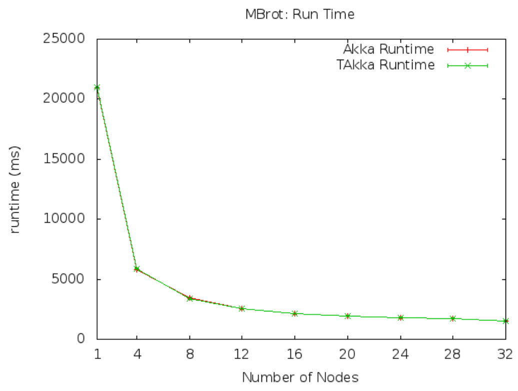
Figure 5.12: Runtime and Efficiency Benchmarks



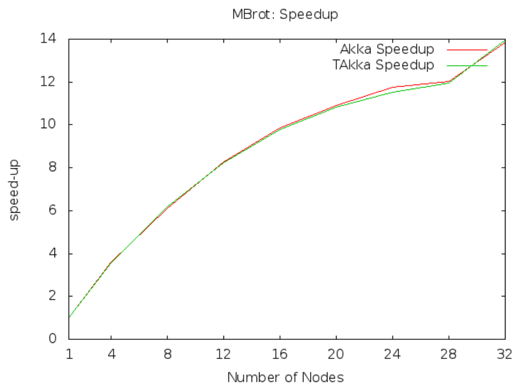
(g) GenStress Time



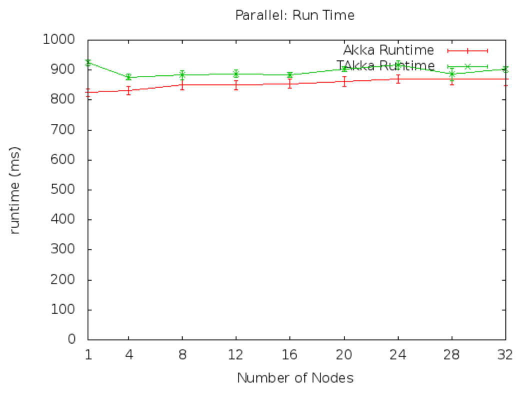
(h) GenStress Scalability



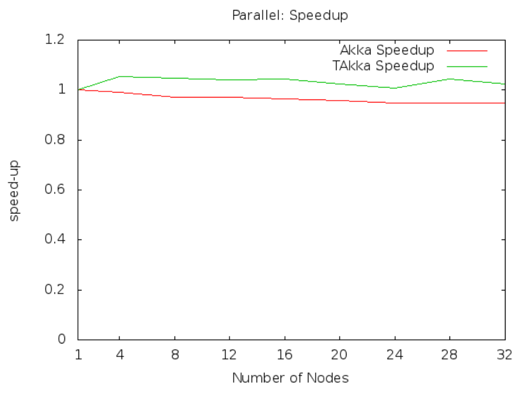
(i) MBrot Time



(j) MBrot Scalability

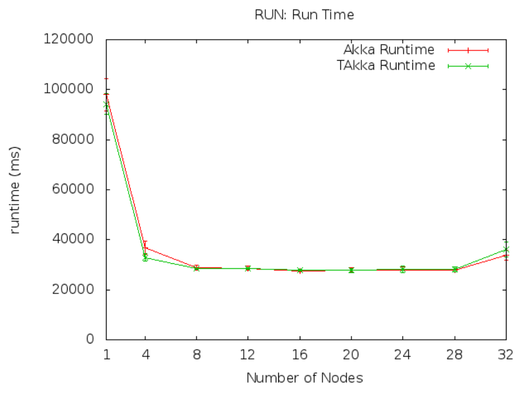


(k) Parallel Time

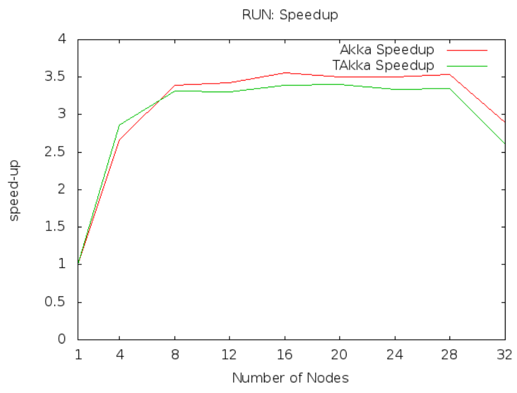


(l) Parallel Scalability

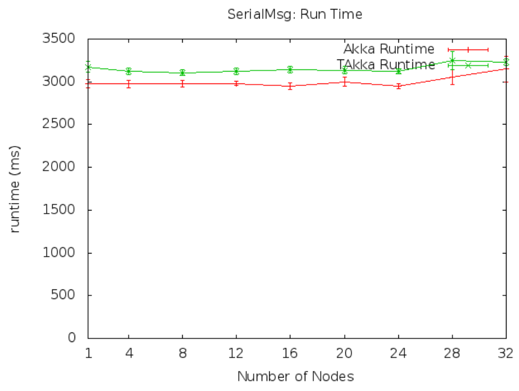
Figure 5.12: Runtime and Efficiency Benchmarks



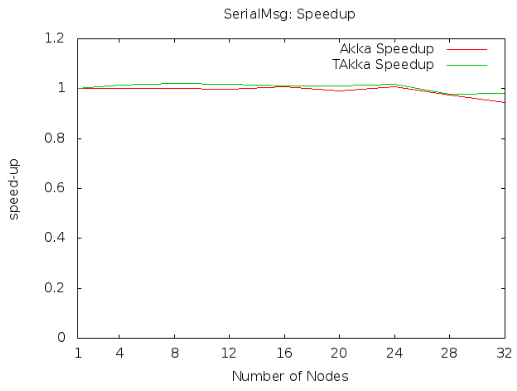
(m) Run Time



(n) Run Scalability



(o) SerialMsg Time



(p) Serial Scalability

Figure 5.12: Runtime and Efficiency Benchmarks

5.4.5 Additional Benchmark: Parallel Fibonacci Numbers

In the last section, we observed that the scalability of BenchErl examples varies. Because BenchErl examples have similar structure and those examples are run on the same environment, we have a reason to believe that the difference of their scalability may lays in the differences of their computational tasks. We expected that the required runtime of a BenchErl example depends on the time for completing the computation task and the time for assembling results. Because the master process is the only process that assembles the results, a BenchErl benchmark is *not* likely to give a good scalability if most of its time is spent on collecting and processing the results of child processes.

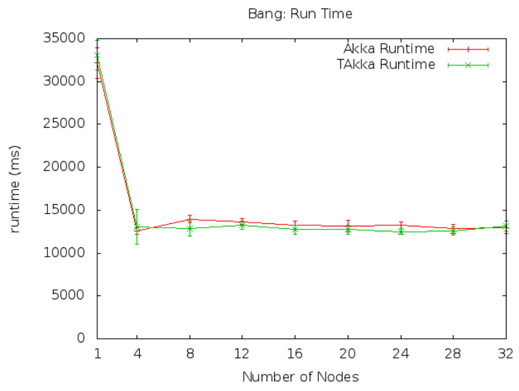
To confirm that the scalability of a BenchErl benchmark depends on the ratio of the time on completing parallelized computational tasks and the time on assembling results, we design and implemented a similar benchmark example where each child process computes a Fibonacci number using the following equation.

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \textit{ or } n = 1 \\ f(n-1) + f(n-2), & \text{if } n >= 2 \end{cases} \tag{5.1}$$

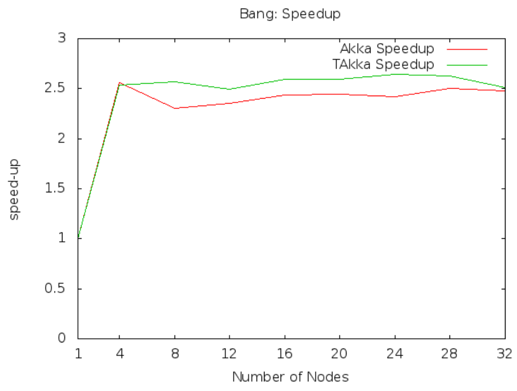
The above basic way of computing a Fibonacci number is chosen because it has a quadratic complexity to the input n, and hence time to compute $f(n)$ changes dramatically when n changes.

The parameters set in this example are the number available nodes, the number child processes, and the value of n in $f(n)$. We expected that, when set the number of child processes to a number that is higher than the number of available nodes, a benchmark with a higher n gives better scalability than those with a lower n.

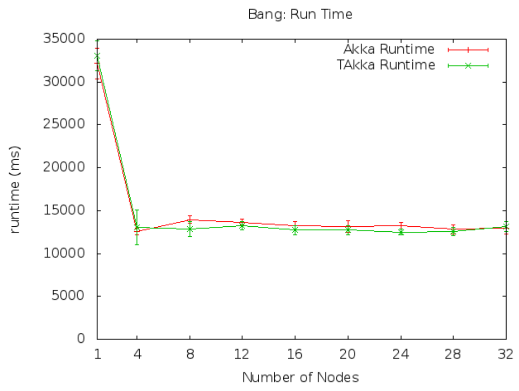
♠**redo the Fibonacci example on Beowulf** ♠ Figure 5.13



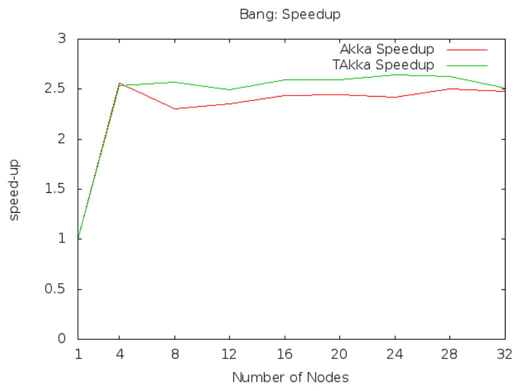
(a) Fib10 Time (To Update)



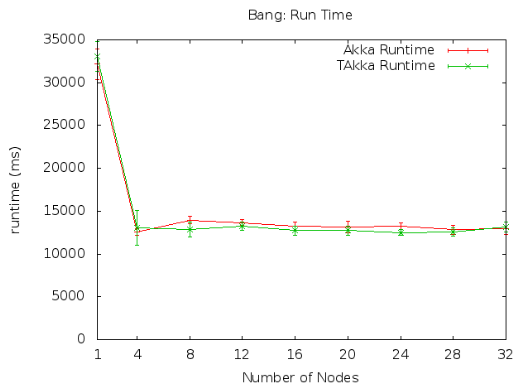
(b) Fib10 Scalability (To Update)



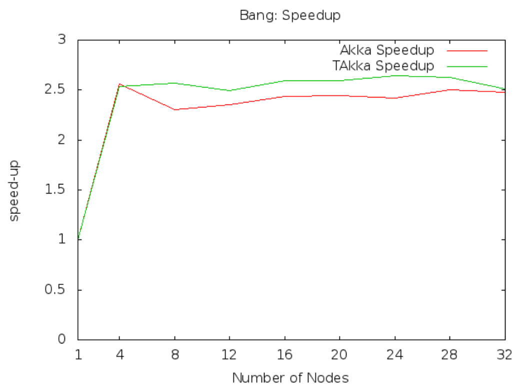
(c) Fib20 Time (To Update)



(d) Fib20 Scalability (To Update)



(e) Fib40 Time (To Update)



(f) Fib40 Scalability (To Update)

Figure 5.13: Benchmark:Parallel Fibonacci Numbers

5.5 Assessing System Reliability and Availability

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 dynamically captures the structure of supervision trees. With the help of Chaos Monkey and Supervision View, users can visualize how their TAkka applications react to adverse conditions. Missing nodes in the supervision tree (Section 5.5.3) show that failures occur during the test. On the other hand, any failed actors are restored, and hence appropriately supervised applications (Section5.5.4) pass Chaos Monkey tests.

Mode	Failure	Description
Random (Default)	Random Failures	Randomly choose one of the other modes in each run.
Exception	Raise an exception	A victim actor randomly raise an exception from a user-defined set of exceptions.
Kill	Failures that can be recovered by scheduling service restart	Terminate a victim actor. The victim actor can be restarted later.
PoisonKill	Unidentifiable failures	Permanently terminate a victim actor. The victim cannot be restarted.
NonTerminate	Design flaw or network congestion	Let a victim actor run into an infinite loop. The victim actor consumes system resources but cannot process any messages.

Table 5.2: 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 5.14: TAkka Chaos Monkey

5.5.1 Chaos Monkey and Supervision View

Chaos Monkey Netflix, Inc. [2013] randomly kills Amazon Elastic Compute Cloud (Amazon EC2) instances in an Auto Scaling Group. In a Chaos Monkey test, the reliability of an application is tested against an intensive adverse conditions. The same idea is ported into Erlang to detect potential flaws of supervision trees Luna [2013]. We port the Erlang version of Chaos Monkey into the TAkka library. In addition to randomly killing actors, users can simulate other common failures by using other modes in Table 5.2.

Figure 5.14 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 5.2. `PoisonPill` and `Kill` are handled by `systemMessageHandler` and can be overrided as described in Section 3.10. `ChaosException` and `ChaosNonTerminate`, on the other hand, are handled by the TAkka library and cannot be overrided.

5.5.2 Supervision View

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

A view master is initialized by calling one of the apply method of the `ViewMaster` object as given in Figure 5.15. 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.

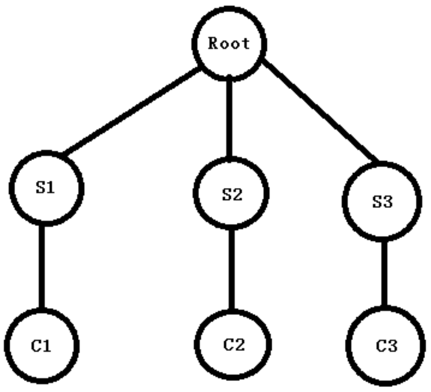
When a TAkka actor receives `SupervisionViewRequest` message, it sends a `SupervisionViewResponse` message back to the view master and pass the `SupervisionViewRequest` message to its children. The `date` value in a `SupervisionViewRespo` message is the same as the `date` value in the corresponding `SupervisionViewRequest` 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.

```
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
}
```

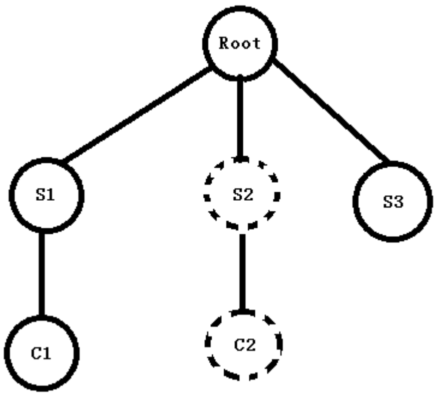
Figure 5.15: Supervision View

5.5.3 A Partly Failed Safe Calculator

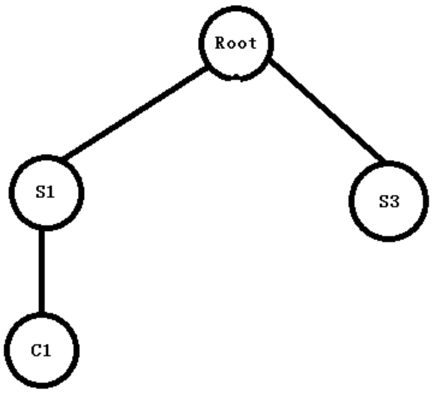
In the hope that Chaos Monkey and Supervision View tests can reveal breaking points of a supervision tree, we modify the Safe Calculator example and run a test as follows. Firstly, we run three safe calculators on three Beowulf nodes, under the supervision of a root actor using the `OneForOne` strategy 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



(a)



(b)



(c)

Figure 5.16: Supervision View Example

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 5.16, gives the expected tree structure at the beginning, 15 seconds and 30 seconds of the test. Figure 5.16a shows that the application initialized three safe calculators as described. In Figure 5.16b, 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 5.16a, the child of S3 is not shown in Figure 5.16b and Figure 5.16c 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 5.16c. 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.

5.5.4 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.

5.6 Summing Up

This chapter confirms that TAkka can detect type errors without bringing in obvious overheads. First, all small and medium sized Akka examples used in this chapter are straightforwardly rewritten using the TAkka library, by updating about 7.4% of the source code. Through the porting process, a type error is found in the Socko framework. We also show that TAkka has its advantage to solve the Wadler's type pollution problem. Second, web servers built using Akka and TAkka reaches similar throughput when the same number of EC2 instances are used. Third, BenchErl benchmark examples written in Akka and TAkka have a similar efficiency and scalability when running on a 32 node Beowulf cluster. The additional benchmark example in Section 5.4.5 gives an evidence that the scalability of an application depends on the ratio of the cost of parallelized computational tasks and the cost of I/O bounded communications. Lastly, we provides a ChaosMonkey library and a Supervision View library for assessing the correctness of applications built using TAkka.

Chapter 6
Future Work

The work presented in this thesis confirms that actors in supervision trees can be typed by parameterising the actor class with the type of messages it expects to receive. The results of our primary evaluations show that the TAkka library can prevent some errors without bringing obvious overheads compared with equivalent Akka applications. As actors and supervision tree are widely used in the development of distributed applications nowadays, we believe that there is a great potential for the TAkka library. Much more can be done to make the TAkka more usable, as well as to further the goal of making the building of reliable distributed applications easier.

6.1 Supervision and Typed Actor in Other Systems

The result of this thesis confirms the feasibility of using type parameterized actors in a supervision tree. The result TAkka library is built on top of Akka for the following three considerations. Firstly, both actor and supervision have been implemented in Akka. The legacy work done by Akka developers makes it possible for us to focus on the core research question. That is, to what extent can actors in a supervision tree be statically typed? Secondly, Akka is built in Scala, a language that has a flexible type system. The flexibility provided by Scala allow us to explore types in a supervision tree. In TAkka, dynamic type checking is only used when static type checking meets its limitations. Thirdly, Akka is a popular programming framework. As part of the typesafe stack, Akka has been used for developing applications in different sizes and for different purposes. If Akka applications can be gradually upgraded to TAkka applications, we believe that the type checking feature in TAkka can improve the reliability of existing Akka systems.

Actor programming has been ported to many languages. The notion of type parameterized actors, however, was introduced very recently in libraries such as Cloud Haskell [Epstein et al., 2011] and scalaz [WorldWide Conferencing, LLC, 2013]. It has been proposed to implement a supervision tree in Cloud Haskell [Watson et al., 2012]. We believe that the techniques used in this thesis can help the design of the future version of Cloud Haskell.

6.2 Benchmark Results from Large Real Applications

This thesis compares TAkka with Akka regarding to several dimensions by porting small and medium sized applications. Most of our examples are selected from open source projects. The author gratefully acknowledges the RELEASE team for giving us access to the source code of the BenchErl benchmark examples. Thomas Arts from Quivq.com and Francesco Cesarini from Erlang Solutions for providing the Erlang source code for the ATM simulator example and the Elevator Controller example, both of which are used in their commercial training courses. Nevertheless, experiments on large real applications are not considered in our evaluation due to the restriction of time and other required resources. It would be interesting to know whether TAkka can help the construction and reliability of large commercial or research applications.

6.3 Supervision Tree that Supports Software Rejuvenation

The core idea of the Supervision Principle is to restart components *reactively* when they fail. Similarly, software rejuvenation [Huang et al., 1995; Dohi et al., 2000] is a *preventive* failure recovery mechanism which periodically restarts components with a clean internal state. The interval of restarting a component, called *software rejuvenate schedule*, is set to a fixed period. Software rejuvenation has been implemented in a number of commercial and scientific applications to improve their longevity. As a supervisor can restart its children, can *software rejuvenate schedule* be set for each actor?

6.4 Measuring and Predicting System Reliability

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

The first approach is measuring the target system as black-box. Unfortunately, Littlewood and Strigini [1993] show that even long time failure-free observation itself does not mean that the software system will achieve a high reliability in the future.

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

- What are possible dependencies between nodes? For each dependency, what is the algebraic relationship between the reliability of a sub-tree and reliabilities of individual nodes?

- Based on the above result, how are the overall reliabilities of a supervision tree to be calculated? When will the reliability be improved by using a supervising tree, and when will not be improved?

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

Chapter 7

Thesis Summary

The main goal of this thesis is the development of a library that combines the advantages of type checking and the supervision principle. Our aim is to contribute to the construction of reliable distributed applications via using type-parameterized actors and supervision trees. Aside from the TAkka library itself, we presented the evaluation results of TAkka. The evaluation metrics in this thesis can be used and further developed for the evaluation of other libraries that implement actors and supervision trees. In this chapter, we first review the research results presented in the thesis and then briefly conclude.

7.1 Overview of Contributions

7.1.1 A library for Type-parameterized Actors and Their Supervision

The key contribution of this thesis is the design and implementation of the TAkka library, which is the first programming library where type parameterized actors can form a supervision tree. The TAkka library is built on top of Akka, a library which has been used for implementing real world applications. The TAkka library adds type checking features to the Akka library but delegates tasks such as actor creation and message passing to the underlying Akka systems.

The TAkka library uses both static and dynamic type checking so that type errors are detected at the earliest opportunity. To enable looking up on remote actor references, TAkka defines a typed name server that keeps maps from typed symbols to values of the corresponding types.

In addition, Akka programs can gradually migrate to their TAkka equivalents (evolution) rather than require providing type parameters everywhere (revolution). The above property is analogous to adding generics to Java programs.

7.1.2 A Library Evaluation Framework

The second contribution of this thesis is a framework for evaluating the TAkka library. We believe that the employed evaluation metrics can be used and further developed for evaluating other libraries that implement actors and the supervision principle.

Compared with Akka, the TAkka library avoids the Wadler's type pollution problem straightforwardly. The type pollution problem, discussed in Section 5.1, refers to the situation where a user can send a service message not expected from him because that service publishes too much type information about its communication interface. Without due care, the type pollution problem may occur in actor-based systems that are constructed using the layered architecture [Dijkstra, 1968; Buschmann et al., 2007] or the MVC pattern [Reenskaug, 1979, 2003], two popular design patterns for constructing modern applications. Our demonstration example shows that avoiding Wadler's type pollution problem in TAkka is as simple as publishing a service as having different types when it is used by different parties.

By porting existing small and medium sized Erlang and Akka applications, results in Section 5.2 and 5.4 show that rewriting Akka programs using TAkka will *not* bring obvious runtime and code-size overheads. As regards expressiveness, *all* Akka applications considered in this thesis can be ported to their TAkka equivalents with a small portion of code modifications. We believe that our TAkka library has the *same* expressiveness as the Akka library.

Finally, the reliability of a TAkka application can be partly assessed by using the Chaos Monkey library and the Supervision View library. The Chaos Monkey library, ported from the work by Netflix, Inc. [2013] and Luna [2013], tests whether an application can survive in an adverse environment where exceptions raise randomly. The Supervision View library dynamically captures the structure of supervision trees. With the help of the Chaos Monkey library and the Supervision View library, application developers can visualise how the application will behave under the tested condition.

7.2 Conclusion

We believe that the demands for distributed applications will continue increasing in the next few years. The recent trends of emphasis on programming for the cloud and mobile platforms all contribute to this direction. With the growing demands and complexity of distributed applications, their reliability will be one of the top concerns among application developers.

The TAkka library introduces a type-parameter for actor-related classes. The additional type-parameter of a TAkka actor specifies the communication interface of that actor. We are glad to see that type-parameterized actors can form supervision trees in the same way as untyped actors. Lastly, test results show that building type-parameterized actors on top of Akka does not introduce significant overheads, with respect to program size, efficiency, and scalability. In addition, debugging techniques such as Chaos Monkey and Supervision View can be applied to applications built using actors with supervision trees. The above results encourage the use of types and supervision trees to implement reliable applications and improve the reliability of legacy applications with little effort. We expect similar results can be obtained in other actor libraries.

Bibliography

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

Allen, R., Lo, N., and Brown, S. (2009). *Zend Framework in action*.

Amazon.com, Inc. (2012). Best practices in evaluating elastic load balancing. http://aws.amazon.com/articles/1636185810492479. Accessed on Oct 2013.

Amazon.com, Inc. (2013a). Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. Accessed on Oct 2013.

Amazon.com, Inc. (2013b). Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing/. Accessed on Oct 2013.

Apple Inc. (2012). Concepts in objective-c programming. [Online; accessed 12-November-2013].

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.

Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., and Venetis, I. E. (2012). A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 33–42. ACM.

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.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. (2006). *The DaCapo Benchmarks: Java benchmarking development and analysis (extended version)*. Department of Computer Science, Faculty of Engineering and Information Technology, Australian National University.

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*.

Buschmann, F., Henney, K., and Schimdt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John Wiley & Sons.

Cesarini, F. (2011). Small to medium applications built with OTP. private communication.

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

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dijkstra, E. W. (1968). The structure of the ''the''-multiprogramming system. *Commun. ACM*, 11:341–346.

Dohi, T., Goseva-Popstojanova, K., and Trivedi, K. S. (2000). Statistical nonparametric algorithms to estimate the optimal software rejuvenation schedule. In *Dependable Computing, 2000. Proceedings. 2000 Pacific Rim International Symposium on*, pages 77–84. IEEE.

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). Dialyzer Reference Manual (version 2.6.1). `http://www.erlang.org/doc/getting_started/users_guide.html`. Accessed on Oct 2013.

Ericsson AB. (2013b). Erlang Reference Manual User's Guide (version 5.10.3). `http://www.erlang.org`. 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.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150.

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

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.

HE, J. (2013). Freebench. https://github.com/Jiansen/FreeBench. Accessed on Oct 2013.

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*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. (1995). Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390. IEEE.

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.

JSON ORG (2013). Introducing JSON. http://json.org. 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.

Lindahl, T. and Sagonas, K. (2004). Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Programming Languages and Systems*, pages 91–106. Springer.

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

Luna, D. (2013). Erlang Chaos Monkey. `https://github.com/dLuna/chaos_monkey`. Accessed on Mar 2013.

Marlow, S. and Wadler, P. (1997). A practical subtyping system for erlang. *ACM SIGPLAN Notices*, 32(8):136–149.

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.

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

Odersky., M. (2013). The Scala Language Specification 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.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Reenskaug, T. (2003). The Model-View-Controller (MVC): Its Past and Present.

Reenskaug, T. M. H. (1979). The original MVC reports.

Sabelfeld, A. and Mantel, H. (2002). Securing communication in a concurrent language. In *Proceedings of the 9th International Symposium on Static Analysis*, pages 376–394. Springer-Verlag.

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.

Typelevel ORG (2013). scalaz: Functional programming for Scala. `http://typelevel.org/projects/scalaz/`.

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.

Watson, T., Epstein, J., Jones, S. P., and He, J. (2012). Supporting libraries (a la otp) for cloud haskel. private communication.

Wikipedia (2013a). Dining philosophers problem. [Online; accessed 12-November-2013].

Wikipedia (2013b). Mandelbrot set. [Online; accessed 12-November-2013].

Wikipedia (2013c). Sleeping barber problem. [Online; accessed 12-November-2013].

Wikipedia (2013d). Tic-tac-toe. [Online; accessed 12-November-2013].

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

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

Zhang, Y. (2008). Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c. Accessed on Oct 2013.