THE UNIVERSITY *of* EDINBURGH

Type-parameterized Actors and Their Supervision

Jiansen HE

Master of Philosophy
Department of Computer Science
University of Edinburgh
2014

Acknowledgements

This thesis is dedicated to my parents for their endless love and support. It is also dedicated to my wife, Shiye, for her continuous encouragement during my hard times. Meeting my wife and getting married are the best two things happened to me in the past 4 years.

I want to thank my supervisors, Philip Wadler, Philip Trinder and Don Sannella, for their guidance over 4 years. I am fortunate to have been supervised by them.

I want to thank my examiners, Ian Stark and Kevin Hammond, for their meticulous examination and detailed suggestions that contribute to the final version of this thesis.

I gratefully acknowledge the substantial help that have received from many colleagues who have shared their related results and ideas over the long period during which this thesis was in preparation. Benedict Kavanagh and Danel Ahman for continuous comments and discussions. Roland Kuhn from the Akka team for sharing deep insight of the the Akka design. The RELEASE team for giving us access to the source code of the BenchErl benchmark examples. Thomas Arts from QuviQ.com and Francesco Cesarini from Erlang Solutions for providing the Erlang source code of two examples used in their commercial training courses.

Declaration

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

(*Jiansen HE*)

Table of Contents

Abstract

The robustness of actor-based concurrent applications can be improved upon by (i) employing failure recovery mechanisms such as the supervision principle, or (ii) using typed messages to prevent ill-typed communication. This thesis explores to what extent the supervision principle can work with typed messages. This thesis evaluates the hypothesis by building a new library called TAkka in the Scala language on top of the existing Akka library, where Akka provides supervised actors and TAkka adds typed messaging. The TAkka library mixes static and dynamic type checking to make sure that dynamically typed distributed resources and statically typed local resources have consistent types. Our notion of typed actor can publish itself as different types when used by different parties so that messages of unexpected types are prevented at the senders' side. In TAkka, messages for supervision purposes are treated in a special way so that a supervisor can interact with child actors of different types. This thesis evaluates the TAkka library by porting 23 small and medium sized Akka applications to their TAkka equivalents. Results show that Akka programs can be gradually upgraded to TAkka equivalents with minimal runtime and code size overheads. Finally, TAkka includes two auxiliary libraries for reliability assessment. This thesis confirms that the supervision principle and typed messages can be merged in an actor library for building real world applications.

Lay Summary

A "programmer" specifies a task for a computer to perform by encoding a sequence of instructions known as a "program" or an "application". Many modern computer application involve programs concurrently executed on one or more computers. Writing a reliable concurrent computer application is challenging to programmers due to its complexity. In practice, two methods can help the robustness of a program. The first method is to use type checking. Types specify how data should be used in programs. Type checking examines if data is used in the right manner in a program. Static type checking reports errors before a program is executed whereas dynamic type checking reports errors when it is executed. Engineering experience shows that errors are easier to be fixed if they were noticed earlier. The other method is to use a library, a collection of programs written and tested by other programmers for common programming tasks. Typically, a library encourages writing programs in certain manners. This thesis interests in a library called "Akka" which encourages "actor programming" and "supervision principle". In actor programming, concurrent programs are coded as actors, which independently perform their own tasks and collaborate by sending messages to each other. The "supervision principle" requires that actors should be organised in a tree structure so that the failure of an actor can be recovered by its supervisor. Unfortunately, the Akka library does not check the type of messages sending to actors, resulting in potential errors otherwise can be avoided.

This thesis presents the result of improving the Akka library by building a TAkka library built on top of the former. The TAkka library employs static type checking whenever applicable. and use dynamic type checking when static checking meets its limitation. It confirms that static type checking and the supervision principle works well together. Apart from the robustness gained from the supervision principle, static type checking result in additional benefits. Firstly, format of messages sent between distributed machines are checked at the earliest possibility. Secondly, typed programs are usually better structured, shorter and easier to maintain. Thirdly, a statically typed program can often be compiled to code that executes more quickly.

This thesis evaluates the TAkka library by porting 23 small and medium sized Akka applications to their TAkka equivalents. Results show that Akka programs can be gradually written to TAkka equivalents with minimal runtime and code size overheads. Finally, TAkka includes two auxiliary libraries for reliability assessment.

Chapter 1

Introduction

This introductory chapter presents the general background that motivates solving problems discussed in this thesis. It summarizes main contributions of this thesis. Finally, an overview of the thesis is given.

1.1 General Background and Motivation

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 API 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. This research project is motivated by the following benefits of statical typing:

1. Both users and developers of actor-based services can take advantages of type-parameterized actors. 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. Another immediate benefits of typing checking is pattern completeness checking for message handlers.

2. Static typing enforces disciplined and modular programming [Pierce, 2002]. Opposite to writing programs in dynamically typed languages, programers are often more confident to use complex and deep nested types in statically typed languages. Using complex types that have a higher level of abstraction improves the readability and maintainability of programs. On the other side, porting nested types into an untyped system using nested tuples is possible but often results in complex code that is more difficult to understand.

3. Static typing often results in shorter code [Pierce, 2002]. When a new actor is defined, handlers for ill-typed messages are no longer needed. When an application is built on top of some actors, which are often black boxes to application developers, there is no need to study how those actors will behave upon receiving unexpected messages and implement handlers for every problematic cases.

4. A statically typed program can often be compiled to code that executes more quickly [Pierce, 2002]. As the compiler knows the exact data types that are in use at runtime, ad hoc optimizations might be applied to the assembly code or the machine code.

5. Statically typed interface is a clear and precise documentation per se [Pierce, 2002]. Users of a type-parameterized actor understands its functionality immediately by looking at its type parameter, which denotes the type of permitted messages. Without the type information, users

would be more rely on good documentation, examples, and even small experiments created by themselves[Endrikat et al., 2014].

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 API 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?

1.2 Contributions

The overall goal of the thesis is to develop a framework that makes it possible to construct reliable distributed applications written using and validated by our library, TAkka, which merges the advantages of type checking and the supervision principle.

The TAkka library is implemented in the Scala programming language. It expands the existing Akka library for supervised actors by the introduction of

typed messaging. Akka is developed at TypeSafe Inc. As Akka becomes part of the standard library in Scala 2.10 and higher versions, it is widely used in many real applications. Choosing Akka as the base of our design and implementation benefits this project in many aspects. Firstly, the author checks that main requirements in distributed programming are not unintentionally neglected by rewriting 23 Akka applications written by other professional programmers. Secondly, improvements made in the TAkka library can benefit existing Akka applications with a low cost because, as presented later, rewriting Akka applications using TAkka involves less than 10% straightforward code changes. Finally, despite some deficits in the Akka design, it saves the author a significant amount of work on low-level implementation. In fact, some less carefully designed Akka features have inspired the author to make improvements in the TAkka library.

Source code of the TAkka library, testing and demonstration examples, along with other related documentations produced during this research are available at a GitHub repository for this project [HE, 2014a] and the CD attached to this thesis. A condensed version of the material in this thesis is published in a companion paper [He et al., 2014].

Generally speaking, the key contributions of this thesis are:

- The design and implementation of the TAkka library (Chapter 3), where supervised actors are parameterized by the type of messages they expect. The library 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. Chapter 4 shows that Akka programs can be upgraded to their TAkka equivalents incrementally, one module at a time (evolution), rather than requiring a monolithic change to all modules simultaneously (revolution). The design is analogous to a design principle of Java Generics, known as "Evolution, not Revolution"

- A framework for evaluating libraries that supports the supervision principle. Chapter 5 shows that the type pollution problem can be straightforwardly avoided in TAkka. The evaluation further 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, TAkka ports 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.

1.3 Thesis Outline

The rest of this thesis is structured as the followings.

Chapter 2 summarises work that influences the design and implementation of the TAkka library. It introduces elements of the Actor Programming model and the Supervision principle, together with short explanations of their usages in the Erlang language and the Akka library. Chapter 2 concludes with features of the Scala type system used in the TAkka implementation.

A condensed version of the material in Chapter 3 to 5 appears in a companion paper. The paper [He et al., 2014] is written as a brief introduction to the TAkka library. It is structured in a way that make the comparison of TAkka and Akka easy for Scala programmers. This thesis elaborates the rational for the TAkka design and implementation. Chapter 3 presents the design and implementation of the TAkka library. Chapter 4 shows that Akka programs can be rewritten using TAkka incrementally, one module at a time. Chapter 5 evaluates the TAkka library.

Chapter 6 concludes and suggests future work that can help the construction of reliable distributed applications.

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 (Section 2.1) and the Supervision principle (Section 2.2), then explains OTP design principles in Erlang (Section 2.3), followed by a short tutorial on how to use the Actor programming model and the Supervision principle in the Akka library (Section 2.4). The Chapter concludes with a summary of features of the Scala type system used in the TAkka implementation (Section 2.5). The Actor model makes concurrent programming easy. The Supervision principle makes applications robust. The Supervision principle is introduced in the Erlang language. It is 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 was 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 either. 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 proto-

type 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 was first proposed in the Erlang/OTP library [Ericsson AB., 2013c] and was 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 provides fault-tolerant support for many enterprise-level distributed real-time applications, which often contain components implemented using other languages. One of the early OTP applications, Ericsson's 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.

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`. In Erlang API, the signature of a function contains its name and the number of arguments it takes. Because Erlang is dynamically typed, users of an Erlang library need to study the type of each function from respected documentation. 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 API 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, Table 2.1 summarises an analogy between Erlang/OTP design principles and common practices in Java and Scala programming.

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

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, or using Java/Scala an equivalent class such as `Thread`.
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

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

Second, since the Erlang application module is just a special behaviour, a programmer 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, the author 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 a Scala library that enforces the supervision principle. The next section briefly introduces Scala features that used in this thesis. 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]. The Akka API used in this thesis is listed in Figure A.1, Appendix A.

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. Analogous 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.

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.

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

```
1 package sample.akka
2
3 import akka.actor.{Actor, ActorRef, ActorSystem, Props}
4
5 class StringCounter extends Actor {
6   var counter = 0;
7   def receive = {
8     case m:String =>
9       counter = counter +1
10      println("received "+counter+" message(s):\n\t"+m)
11  }
12 }
13
14 class MessageHandler extends Actor {
15   def receive = {
16     case akka.actor.UnhandledMessage(message, sender, recipient) =>
17       println("unhandled message:"+message);
18   }
19 }
20
21 object StringCounterTest extends App {
22   val system = ActorSystem("StringCounterTest")
23   val counter = system.actorOf(Props[StringCounter], "counter")
24
25   val handler = system.actorOf(Props[MessageHandler]))
26   system.eventStream.subscribe(handler,classOf[akka.actor.UnhandledMessage]);
27   counter ! "Hello World"
28   counter ! 1
29   Thread.sleep(1000)
30   val counterRef =
       system.actorFor("akka://StringCounterTest/user/counter")
31   counterRef ! "Hello World Again"
32   counterRef ! 2
33 }
34
35
36 /*
37 Terminal output:
38 received 1 message(s):
39   Hello World
40 unhandled message:1
41 received 2 message(s):
42   Hello World Again
43 unhandled message:2
44 */
```

Figure 2.1: Akka Example: A String Counter

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 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 the central topics of this thesis.

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.

```
1 trait Actor extends AnyRef
2   type Receive = PartialFunction[Any, Unit]
3
4   abstract def receive: Actor.Receive
5   implicit final val self: ActorRef
6   implicit val context: ActorContext
7   def supervisorStrategy: SupervisorStrategy
8
9   final def sender: ActorRef
10
11  def preStart(): Unit
12  def preRestart(reason: Throwable, message: Option[Any]): Unit
13  def postRestart(reason: Throwable): Unit
14  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 decided that the sender of the last message should 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]. Alternatives to working with mutable states will be discussed in Section 3.13.

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 actually has type `Function[String, Unit]` rather than the declared type `PartialFunction[Any, Unit]`. The definition of `StringCounter` is accepted by the Scala 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 presumably 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 API 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.

```
1 package akka.actor
2 final case class Props(deploy: Deploy, clazz: Class[_],
3                        args: Seq[Any]) extends Product with Serializable
4
5 object Props extends Serializable
6   def apply[T <: Actor]()(implicit arg0: ClassManifest[T]): Props
7   def apply(clazz: Class[_], args: Any*): Props
```

Figure 2.3: Akka API: Props

An example of creating a `Props` instance is given in Figure 2.1, that is:

```
1 Props[StringCounter]
```

which is short for

```
1 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(classOf[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` (Section 2.4.1.1) or `ActorContext` (Section 2.4.1.6). In Figure 2.1, `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 supervision will be given in Section 2.4.2.

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") .
```

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
```

```
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

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 specified 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. In contrast, 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 mentioned a few times in previous sections. This section explains what the contextual information of an Akka actor includes, with a reference to the following API cited from [Typesafe Inc. (a), 2012].

The API in Figure 2.5 shows two groups of methods: those for interacting

```
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

with other actors (lines 3 to 24), and those for controlling the behaviour of the 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, among all considered examples, messages are sent to specific actors rather than a tree of actors.

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, the author believes 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, `setReceiveTimeout`, specifies the timeout within which a new message shall be received; otherwise a `ReceiveTimeout` message is sent to the actor. The second behaviour, *receive*, is the handler for incoming messages. The next subsection explains how to upgrade the message handler of an actor using the `become` and `unbecome` method.

2.4.1.7 Dynamic Behaviour Upgrade

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

```
1 package sample.akka
2 import akka.actor._
3 case object Upgrade
4 case object Downgrade
5 case class Mul(m:Int, n:Int)
6 case class Div(m:Int, n:Int)
7 class CalculatorServer extends Actor {
8   import context._
9   def receive = simpleCalculator
10  def simpleCalculator:Receive = {
11    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
12    case Upgrade =>
13      println("Upgrade")
14      become(advancedCalculator, discardOld=false)
15    case op =>    println("Unrecognised operation: "+op)
16  }
17  def advancedCalculator:Receive = {
18    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
19    case Div(m:Int, n:Int) =>  println(m +" / "+ n +" = "+ (m/n))
20    case Downgrade =>
21      println("Downgrade")
22      unbecome()
23    case op =>    println("Unrecognised operation: "+op)
24 } }
25 object CalculatorUpgrade extends App {
26    val system = ActorSystem("CalculatorSystem")
27    val calculator:ActorRef = system.actorOf(Props[CalculatorServer],
          "calculator")
28   calculator ! Mul(5, 1)
29   calculator ! Div(10, 1)
30   calculator ! Upgrade
31   calculator ! Mul(5, 2)
32   calculator ! Div(10, 2)
33   calculator ! Downgrade
34   calculator ! Mul(5, 3)
35   calculator ! Div(10, 3)
36 }
37 /*  Terminal output:
38 5 * 1 = 5
39 Unrecognised operation: Div(10,1)
40 Upgrade
41 5 * 2 = 10
42 10 / 2 = 5
43 Downgrade
44 5 * 3 = 15
45 Unrecognised operation: Div(10,3)
46  */
```

Figure 2.6: Akka Example: Behaviour Upgrade

calculator starts with a basic version that can only compute multiplication. When it receives an `Upgrade` command, it upgrades to an advanced version that can compute both multiplication and division. The advanced calculator downgrades to the basic version when it receives a `Downgrade` command.

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 restarts a child when it fails. The `AllForOne` supervisor strategy corresponds to the `one_for_all` supervision strategy in OTP, which restarts 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.

```
1 package akka.actor
2 abstract class SupervisorStrategy
3 case class OneForOne(restart:Int, time:Duration)(decider: Throwable =>
4   Directive) extends SupervisorStrategy
5 case class AllForOne(restart:Int, time:Duration)(decider: Throwable =>
6   Directive) extends SupervisorStrategy
7
8 sealed trait Directive extends AnyRef
9 object Escalate extends Directive
10 object Restart extends Directive
11 object Resume extends Directive
12 object Stop extends Directive
```

Figure 2.7: Akka API: Supervisor Strategies

Figure 2.7 gives the API 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. The code then defines a safe calculator as the supervisor of the simple calculator. The

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

Figure 2.8: Akka Example: Supervised Calculator

safe calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` is raised. A supervisor logs exceptions raised from its children by default. In this example, logs are printed to the terminal. 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. A more robust application should react to failures of all user created actors. However, an Akka guardian actor keeps silent when its child fails.

2.5 Essential Scala Features

One of the key design principles of the TAkka library, described in subsequent chapters, is using static 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 language that benefit implementation of the TAkka library.

2.5.1 The Scala Language

The Scala programming language merges features of functional programming and object-oriented programming. A Scala program is compiled to Java bytecode and runs on a Java virtual machine (JVM). The syntax of the Scala language is similar to Java. The Scala website [EPFL, 2014] gives Scala tutorials for programmers from different background. This section briefly lists some general features of the Scala language compared with Java. Later sections will describe the Scala type system which is important to the implementation of the TAkka library, the key result of this thesis.

Scala uses a statically typed system. In Scala, all types inherit from a top-level class `Any`. In the extreme case where all values are declared and used as type `Any`, a program is the same as its dynamically typed equivalent. The above property gives programmers an advantage of working in a gradual typing system where a program can be dynamically typed in earlier versions and be migrated to a more statically typed version later. Additionally, Scala has

a sophisticated type inference support on local variables, which often results in clean code.

Scala supports many syntactical sugars and functional features such as currying, pattern matching, partial functions and lazy evaluation. Syntactical sugars makes Scala a good language for certain style of domain-specific languages. The support for functional programming gives programmers the flexibility of writing programs in a style that takes the advantages of both functional and object-oriented style.

2.5.2 Parameterized Types

A *parameterized type* `T[`U_1`,...,`U_n`]` consists of a type constructor `T` and a positive number of type parameters $U_1, \ldots, U_n$ [Odersky., 2013]. The type constructor `T` must be a valid type name whereas each type parameter U_i is a type. Scala Parameterized Types are similar to Java and C# generics and C++ templates, but express *variance* and *bounds* differently as explained later.

2.5.2.1 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 adapted from [Naftalin and Wadler, 2006, Example 5-2]. The example defines an abstract data type `Stack`, an implementation class `ListStack`, a utility method `reverse`, and client application `Client`.

In the example, `Stack` is defined as a *trait*. A Scala *trait* supports inheritance. Different from a Java *Interface*, a *trait* can contain one or more method implementations. Compared with an abstract class, a *trait* supports multiple inheritance. The `Stack` trait defines the signature of three methods: `empty`, `push`, and `pop`. 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 `ListStack` class implements the `Stack` trait using the `List` data structure. Different to Java, Scala classes do not have static members. Therefore, the utility method `reverse` 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. Finally, the `Client` application tests the generic stack. Line 10 of the `Client` test shows that `Stack[Integer]` is not a subtype of

```scala
package sample.scala.generic.mutable
trait Stack[E] {
  def empty():Boolean
  def push(elt:E):Unit
  def pop():E
}
```

```scala
class ListStack[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 ListStack[T]
    while(!in.empty){
      val elt = in.pop
      out.push(elt)
    }
    return out
  }
}
```

```scala
object Client extends App {
  val stack:Stack[Integer] = new ListStack[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)"))
  // var stack2:Stack[Any] = stack // compile error
}
```

Figure 2.9: Scala Example: A Generic Stack Library

`Stack[Any]` in this example. The next section discusses subtyping relationship between generic classes of the same type constructor, known as *variance*.

2.5.2.2 Variance and Type Bounds

An important topic skirted in the last sub-section is how variance under inheritance works in Scala. Specifically, if $\texttt{T}_{sub}$ is a subtype of `T`, is $\texttt{Stack[T}_{sub}\texttt{]}$ the subtype of `Stack[T]`, or the reverse? Unlike Java generics [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. the only subtype of X[T] is itself)

Table 2.2: Variance Under Inheritance

A variance annotation constrains how a type variable can be used. Scala 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 finds 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 covariant type when you only get values out of a structure, use a contravariant type when you only put values into a structure, and use an invariant type when you both get and put.*

```scala
package sample.scala.generic.immutable
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 ListStack[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 ListStack(elt :: list) }
  def pop():(E, Stack[E]) = {
    if (!empty) (list.head, new ListStack(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 ListStack[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 ListStack[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)"))
  }
  var stack2:Stack[Any] = stack
}
```

Figure 2.10: Scala Example: A Covariant Immutable Stack

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 class which is covariant on its type parameter, as shown in Figure 2.10. The immutable.Stack class is defined as 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, as long as the type of Stack is declared as `Stack[+E]`, the signature of its `push` method *cannot* be

```
1 def push[E](elt:E):Unit
```

Otherwise, a user can put a value of type `Any` to a stack of type `Stack[Integer]`, which is a subtype of `Stack[Any]`. An alternative 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. Line 14 the of `Client` test in Figure 2.10 shows that `immutable.Stack[Integer]` is a subtype of `immutable.Stack[Any]` in this example.

In Figure 2.10, the signature of `push` is changed to `push[T>:E](elt:T):Stack[T]`, with an additional type parameter `T>:E` which denotes that `T` is a supertype of `E`. In Scala, `E` is called the *lower bound* of `T`. Similarly, `T<:E` means `T` is a subtype of `E` and `E` is called the *upper bound* of `T`. In Scala, `Any` is the supertype of all types and `Nothing` is the subtype of all types.

2.5.3 Scala Type Descriptors

As in Java, generic types are erased at runtime in Scala. To record type information that is required at runtime, users can ask Scala to keep the type information by using the `Manifest` class. A `Manifest[T]` encapsulates the runtime type representation of some type `T`. It provides methods for subtype test (<:<).

Figure 2.11 gives some examples of using `Manifest`. The example shows that a `Manifest` value (line 9) records the value of a type parameter where as a `Class` (line 10) does not. To define a method that obtains type information of a generic type, the `typeName` defined at line 13 asks the Scala runtime to provide a value of `Manifest`. To simplify the API, Scala further provides a syntactic sugar called context bounds. We define an equivalent method `boundTypeName` using context bounds at Line 18.

```scala
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

  assert(manifest[List[Int]].toString.equals(
          "scala.collection.immutable.List[Int]"))
  assert(classOf[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 2.11: Scala Example: Manifest Example

2.6 Other Related Work

This section summarizes other related work that inspires this research although not directly influence the final result. Section 2.6.1 introduces the Join-Calculus, a computation model where a computation component can have different channels for different messages. Appendix B presents the Scala Join library implemented by the author as an exercise to understand distributed channel-based communication. Section 2.6.2 introduces the Ambient Calculus, an early computation model where code can executed on and migrates between distributed computational components.

2.6.1 The Join-Calculus and the JoCaml Programming Language

Join-calculus [Fournet et al., 1995] is a name passing process calculus that is designed for the distributed programming. There are two versions of the join-calculus, namely the core join-calculus and the distributed join-calculus. The core join-calculus could be considered as a variant of the π-calculus. It is as expressive as the asynchronous π-calculus in the sense that translations between those two calculi are well formulated. A remarkable construct in the join-calculus is join patterns, which provides a convenient way to express process synchronisations. This feature also makes the join-calculus closer to a real programming language. The distributed join-calculus extends the core calculus with location, process migration, and failure recovery. This proposal uses the short phrase "join-calculus" to refer to the distributed join-calculus which includes all components in the core join-calculus. The syntax (Table 2.3), the scoping rules (Table 2.4), and the reduction rules (Table 2.5) of the join-calculus are cited from [Fournet et al., 1996].

P	$::=$		processes	D	$::=$		definition
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message		$\mid$	$J \triangleright P$	local rule
					$\mid$	$\top$	inert definition
	$\mid$	**def** D **in** P	local definition		$\mid$	$D \wedge D$	co-definition
	$\mid$	$P \mid P$	parallel composition		$\mid$	$a\,[D : P]$	**sub-location**
					$\mid$	$\Omega a\,[D : P]$	**dead sub-location**
	$\mid$	**0**	inert process	J	$::=$		join-pattern
	$\mid$	$go\langle a, \kappa\rangle$	**migration**		$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
	$\mid$	$halt\langle\rangle$	**termination**		$\mid$	$J \mid J$	synchronous join-pattern
	$\mid$	$fail\langle a, \kappa\rangle$	**failure detection**				

Constructs whose explanation is in **bold** font are only used in the distributed join-calculus. Other constructs are used in both distributed and local join-calculus.

Table 2.3: Syntax of the distributed join-calculus – [Fournet et al., 1995]

2.6.1.1 The Local Reflexive Chemical Machine (RCHAM)

The denotational semantics of the join-calculus is usually described in the domain of a reflexive chemical machine (RCHAM). A local RCHAM consists of two parts: a multiset of definitions D and a multiset of active processes P. Definitions specify possible reductions of processes, while active processes can introduce new names and reaction rules.

$J:$	$dv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{x\}$	$rv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{u \in \widetilde{v}\}$
	$dv[J \mid J']$	$\stackrel{def}{=}$	$dv[J] \cup dv[J']$	$rv[J \mid J']$	$\stackrel{def}{=}$	$rv[J] \uplus rv[J']$
$D:$	$dv[J \triangleright P]$	$\stackrel{def}{=}$	$dv[J]$	$rv[J \triangleright P]$	$\stackrel{def}{=}$	$dv[J] \cup (fv[P] - rv[J])$
	$dv[\top]$	$\stackrel{def}{=}$	$\emptyset$	$fv[\top]$	$\stackrel{def}{=}$	$\emptyset$
	$dv[D \wedge D']$	$\stackrel{def}{=}$	$dv[D] \cup dv[D']$	$fv[D \wedge D']$	$\stackrel{def}{=}$	$fv[D] \cup fv[D']$
	$dv[a\,[D : P]]$	$\stackrel{def}{=}$	$\{a\} \uplus dv[D]$	$fv[a\,[D : P]]$	$\stackrel{def}{=}$	$\{a\} \cup fv[D] \cup fv[P]$
$P:$	$fv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{x\} \cup \{u \in \widetilde{v}\}$	$fv[go\langle a, \kappa\rangle]$	$\stackrel{def}{=}$	$\{a, \kappa\}$
	$fv[\mathbf{0}]$	$\stackrel{def}{=}$	$\emptyset$	$fv[halt\langle\rangle]$	$\stackrel{def}{=}$	$\emptyset$
	$fv[P \mid P']$	$\stackrel{def}{=}$	$fv[P] \cup fv[P']$	$fv[fail\langle a, \kappa\rangle]$	$\stackrel{def}{=}$	$\{a, \kappa\}$
	$fv[\mathbf{def}\ D\ \mathbf{in}\ P]$	$\stackrel{def}{=}$	$(fv[P] \cup fv[D]) - dv[D]$			

Well-formed conditions for D: A location name can be defined only once; a channel name can only appear in the join-patterns at one location.

Table 2.4: Scopes of the distributed join-calculus – [Fournet et al., 1995]

The six chemical rules for the local RCHAM are **str-join**, **str-null**, **str-and**, **str-nodef**, **str-def**, and **red** in Table 2.5. As their names suggest, the first 5 are structure rules whereas the last one is reduction rule. Structure rules correspond to reversible syntactical rearrangements. The reduction rule, **red**, on the other hand, represents an irreversible computation.

Finally, for the ease of writing programs, the local join-calculus could be extended with synchronous channel, sequencing, and let-bindings as in Table 2.6. The distributed join-calculus could be extended similarly.

2.6.1.2 Distributed Solutions

Distributed system in the join-calculus is constructed in three steps: first, definitions and processes are partitioned into several local solutions; then, each local solution is attached with a unique location name; finally, location names are organized in a location tree.

A distributed reflexive chemical machine (DRCHAM) is simply a multiset of RCHAMs. It is important to note that message pending to a remotely defined channel will be forwarded to the RCHAM where the channel is defined before applying any **red** rule. The above process is a distinction between the join-calculus and other distributed models. The side effect of this evaluation strategy is that both channel and location names must be pairwise distinct in the whole system. As a consequence, a sophisticate name scheme is required

str-join	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1,\ P_2$	
str-null	$\vdash \mathbf{0}$	$\rightleftharpoons$	$\vdash$	
str-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1,\ D_2 \vdash$	
str-nodef	$\top$	$\rightleftharpoons$	$\vdash$	
str-def	$\vdash \mathbf{def}\ D\ \mathbf{in}\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$	(range(σ_{dv}) fresh)
str-loc	$\varepsilon a\ [D : P] \vdash_\varphi$	$\rightleftharpoons$	$\vdash_\varphi \parallel \{D\} \vdash_{\varphi\varepsilon a} \{P\}$	(a frozen)
red	$J \triangleright P \vdash_\varphi J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash_\varphi P\sigma_{rv}$	(φ alive)
comm	$\vdash_\varphi x\langle\widetilde{v}\rangle \parallel J \triangleright P \vdash$	$\longrightarrow$	$\vdash_\varphi \parallel J \triangleright P \vdash x\langle\widetilde{v}\rangle$	($x \in dv[J]$, φ alive)
move	$a[D : P\|go\langle b,\kappa\rangle] \vdash_\varphi \parallel \vdash_{\psi\varepsilon b}$	$\longrightarrow$	$\vdash_\varphi \parallel a\ [D : P\|\kappa\langle\rangle] \vdash_{\psi\varepsilon b}$	(φ alive)
halt	$a[D : P\|halt\langle\rangle] \vdash_\varphi$	$\longrightarrow$	$\Omega a[D : P] \vdash_\varphi$	(φ alive)
detect	$\vdash_\varphi fail\langle a,\kappa\rangle \parallel \vdash_{\psi\varepsilon a}$	$\longrightarrow$	$\vdash_\varphi \kappa\langle\rangle \parallel \vdash_{\psi\varepsilon a}$	($\psi\varepsilon a$ dead, φ alive)

Side conditions: in **str-def**, σ_{dv} instantiates the channel variables $dv[D]$ to distinct, fresh names; in **red**, σ_{rv} substitutes the transmitted names for the received variables $rv[J]$; φ is dead if it contains Ω, and alive otherwise; "a frozen" means that a has no sublocations; εa denotes either a or Ωa

Table 2.5: The distributed reflexive chemical machine – [Fournet et al., 1995]

for a language that implements the join-calculus.

To support process migration, a new contract, $go\ \langle b, \kappa\rangle$, is introduced, together with the **move** rule. There are two effects of applying the move rule. Firstly, site a moves from one place (φa) to another ($\psi\varepsilon$a). Secondly, the continuation $\kappa\langle\rangle$ may trigger another computation at the new location.

2.6.1.3 The Failure Model

A failed location in the join-calculus cannot respond to messages. Reactions inside a failed location or its sub-locations are prevented by the side-condition of reduction rules. Nevertheless, messages and locations are allowed to move into a failed location, but will be frozen in that dead location (str-loc).

To model failure and failure recovery, two primitives $halt\langle\rangle$ and $fail\langle\cdot,\cdot\rangle$ are introduced to the calculus. Specifically speaking, $halt\langle\rangle$ terminates the location where it is triggered (rule halt), whereas $fail\langle a, \kappa\rangle$ triggers the continuation $\kappa\langle\rangle$ when location a fails (rule detect).

2.6.1.4 The JoCaml Programming Language

The JoCaml programming language is an extension of OCaml. JoCaml supports the join-calculus with similar syntax and more syntactic sugars. When using JoCaml to build distributed applications, users should be aware of following three limitations in the current release (version 3.12) [Fournet et al., 2003]:

P	$::=$		processes
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message
	$\mid$	**def** D **in** P	local definition
	$\mid$	$P \mid P$	parallel composition
	$\mid$	$\mathbf{0}$	inert process
	$\mid$	$\mathrm{x}(\widetilde{V}); P$	**sequential composition**
	$\mid$	**let** $\widetilde{u} = \widetilde{V}$ **in** P	**named values**
	$\mid$	**reply** $\widetilde{V}$ **to** x	**implicit continuation**
D	$::=$		definition
	$\mid$	$J \triangleright P$	local rule
	$\mid$	$\top$	inert definition
	$\mid$	$D \wedge D$	co-definition
J	$::=$		join-pattern
	$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
	$\mid$	$J \mid J$	synchronous join-pattern
V	$::=$		values
	$\mid$	x	value name
	$\mid$	$\mathrm{x}(\widetilde{V})$	**synchronous call**

$\mathrm{x}(\widetilde{v})$	$=$	$x\langle\widetilde{v}, \kappa_x\rangle$	(in join-patterns)
reply $\widetilde{V}$ **to** x	$=$	$\kappa_x\langle\widetilde{V}\rangle$	(in process)
$x\langle\widetilde{V}\rangle$	$=$	**let** $\widetilde{v} = \widetilde{V}$ **in** $x\langle\widetilde{v}\rangle$	
let $\widetilde{u} = \widetilde{V}$ **in** P	$=$	**let** $u_1 = V_1$ **in let** $u_2 = \cdots$ **in** P	
let $\widetilde{u} = \mathrm{x}(\widetilde{V})$ **in** P	$=$	**def** $\kappa\langle\widetilde{u}\rangle \triangleright P$ **in** $x\langle\widetilde{V}, \kappa\rangle$	
let $u = v$ **in** P	$=$	$P\{v/u\}$	
$\mathrm{x}(\widetilde{V}); P$	$=$	**def** $\kappa\langle\rangle \triangleright P$ **in** $x\langle\widetilde{V}, \kappa\rangle$	

Table 2.6: The core join-calculus with synchronous channel, sequencing, and let-binding – [Fournet et al., 1995]

1. Functions and closures transmission are not supported. In the join-calculus, distributed calculation is modelled as sending messages to a remotely defined channel. As specified in the **comm** rule, messages sent to a remotely defined channel will be forwarded to the place where the channel is defined. In some cases, however, programmers may want to define an computation at one place but execute the computation elsewhere. The standard JoCaml, unfortunately, does not support code mobility.

2. Distributed channels are untyped. In JoCaml, distributed port names are retrieve by enquiring its registered name (a string) from name service. Since JoCaml encourages modular development, codes supposed to be run at

difference places are usually wrote in separated modules and complied independently. The annotated type of a distributed channel, however, is not checked by the name service. Invoking a remote channel whose type is erroneously annotated may cause a run-time error.

3. When mutable value is required over the web, a new copy, rather than a reference to the value, is sent. This may cause problems when a mutable value is referenced at many places across the network.

2.6.2 The Ambient Calculus and the Obliq Programming Language

2.6.2.1 The Ambient Calculus

The ambient calculus provides a formal basis for describing mobility in concurrent systems. Here mobility refers to both *mobile computing* (computation carried out in mobile devices) and *mobile computation* (code moves between the network sites) [Cardelli and Gordon, 1998]. In reality, there is an additional security requirement for mobility, that is, the authorization for an agent to enter or exit certain administrative domain (e.g. a firewall). The ambient calculus solves the above problems with a fundamental concept: ambient. The three key attributes of a ambient are:

- a name for access control (enter, exit, and open the ambient).
- a collection of local processes/agents that control the ambient.
- a collection of sub-ambients.

An atomic computation in the ambient calculus is a one-step movement of an ambient. Although the pure ambient calculus with mobility is Turing-complete [Cardelli and Gordon, 1998], communication primitives are necessary to comfort the encoding of other communication based calculi such as the π-calculus. The full calculus is given through Table 2.7 to 2.9, cited from [Cardelli and Gordon, 1998]. It is important to note that communication in the ambient calculus are local. In other words, value (name or capability) communication only happens between two processes inside the same ambient.

Mobility and Communication Primitives

$P,Q ::=$	processes
$(\nu n)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	capability action
$(x).P$	input action
$\langle M \rangle$	async output action
$M ::=$	capabilities
x	variable
n	name
$in\ M$	can enter into M
$out\ M$	can exit out of M
$open\ M$	can open M
ε	null
$M.M'$	path

Free names (revisions and additions)

$$fn(M[P]) \triangleq fn(M) \cup fn(P) \qquad fn(x) \triangleq \emptyset$$
$$fn((x).P) \triangleq fn(P) \qquad fn(n) \triangleq \{n\}$$
$$fn(\langle M \rangle) \triangleq fn(M) \qquad fn(\varepsilon) \triangleq \emptyset$$
$$fn(M.M') \triangleq fn(M) \cup fn(M')$$

Free variables

$$fv((\nu n)P) \triangleq fv(P) \qquad fv(x) \triangleq \{x\}$$
$$fv(\mathbf{0}) \triangleq \emptyset \qquad fv(n) \triangleq \emptyset$$
$$fv(P \mid Q) \triangleq fv(P) \cup fv(Q) \qquad fv(in\ M) \triangleq fv(M)$$
$$fv(!P) \triangleq fv(P) \qquad fv(out\ M) \triangleq fv(M)$$
$$fv(M[P]) \triangleq fv(M) \cup fv(P) \qquad fv(open\ M) \triangleq fv(M)$$
$$fv(M.P) \triangleq fv(M) \cup fv(P) \qquad fv(\varepsilon) \triangleq \emptyset$$
$$fv((x).P) \triangleq fv(P) - \{x\} \qquad fv(M.M') \triangleq fv(M) \cup fv(M')$$
$$fv(\langle M \rangle) \triangleq fv(M)$$

Table 2.7: Syntax and scope in the ambient-calculus – [Cardelli and Gordon, 1998]

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	(Struct Input)
$\varepsilon.P \equiv P$	(Struct ε)
$(M.M').P \equiv M.M'.P$	(Struct .)

Table 2.8: Structure congurence in the ambient-calculus – [Cardelli and Gordon, 1998]

$n[in\ m.\ P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	(Red Res)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red Amb)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red ≡)
$(x).P \mid \langle M \rangle \rightarrow P\{x \leftarrow M\}$	(Red Comm)

Table 2.9: Reduction in the ambient-calculus – [Cardelli and Gordon, 1998]

2.6.2.2 The Obliq Programming Language

At the time of writing this report, there is no real language that implements the ambient-calculus[1]. Instead, this section will introduce the Obliq language, which has certain notions of ambient and influenced the design of the ambient calculus.

Obliq[Cardelli, 1995] is one of the earliest programming languages which support distributed programming. The language was designed before the pervasive of web applications. It only supports simple object model which is a collection of fields. Each field of an Obliq object could be a value (a constant or a procedure), a method, or an alias to an object. An object could either be constructed directly by specifying its fields, or be cloned from other objects.

The four operations which could be performed on objects are:

- selection: a value field of a object could be selected and transmitted over the web. If the selected value is a constant, the value will be transmitted. By contrast, if the selected value is a method, values of its arguments will be transmitted to the remote site where the method is defined, the computation is performed remotely, and the result or an exception is returned to the site of the selection.

- updating: when an updating operation is performed on an remote object, the selected filed is updated to a value that might be sent across the web. If the selected filed is a method, a transmission of method closure is required.

- cloning: cloning an object will yield a new object which contains all fields of argument objects or raise an error if field names of argument objects conflict.

- aliasing: After executing an aliasing method, a.x := **alias** y **of** b **end**, further operations on x of a will be redirected to y of b.

It is important to note that Obliq, as some other languages in the pre-web era, does not distinguish local values from distributed values. By contrast, Waldo et al. [1997] pointed out that distinct views must be adopted for local and distributed objects, due to differences in latency, memory access, partial failure, and concurrency.

[1]Cruz and Aguirre [2005] proposed a virtual machine for the ambient calculus.

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

A condensed version of the material in this chapter appears
in [He et al., 2014, Section 3 and 4]

The last chapter examined 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.

This chapter presents the design of the TAkka library, which confirms the key claim of this thesis: actors in supervision trees can be *statically* typed by parameterizing the actor class with the type of messages it expects to receive. This chapter outlines 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 on 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, actors can be parameterized as expected. Nevertheless, as the Akka API and the structure of Akka configuration file change slightly in different Akka versions, readers who want to use a later Akka version may need to update the API or the configuration file according to the specification of the Akka version used. The latest TAkka API and its companion Akka API (version 2.1.4) are given in Appendix A.

3.1 TAkka Example: a String Counter

The introduction of TAkka begins 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 equivalent, with a few differences marked in blue.

First, the `Actor` class in TAkka takes a type parameter which 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 the `typedReceive` method is mouseovered:

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

The type of `m` at line 8, which is `String`, is omitted too because it can be inferred as well.

Second, the type of messages sending to an actor reference is statically checked. In the TAkka version of the `StringCounterTest`, the Scala language infers that the type of `counter`, declared at line 16, has type `ActorRef[String]`. This means that only String messages can be sent to `counter`. Sending a non-String message (i.e. line 21) results in a compile error.

Third, 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 23 and 26, TAkka 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 26 is not statically detected, an exception is expected to raise as soon as the ill-typed actor reference is claimed at run time (line 26), earlier than the time when the actor reference is used (line 29). The terminal output shows that the print statement at line 28 has not been executed when the exception is raised. The result confirms that the exception is raised by code at line 26.

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

```
1 package sample.takka
2
3 import takka.actor.{Actor, ActorRef, ActorSystem, Props}
4
5 class StringCounter extends Actor[String] {
6   var counter = 0;
7   def typedReceive = {
8     case m =>
9       counter = counter +1
10      println("received "+counter+" message(s):\n"+m)
11  }
12 }
13
14 object StringCounterTest extends App {
15   val system = ActorSystem("StringCounterTest")
16   val counter = system.actorOf(Props[String, StringCounter],
17 "counter")
18
19   counter ! "Hello World"
20 //  counter ! 1
21 // type mismatch; found : Int(1) required: String
22   val counterString =
23 system.actorFor[String]("akka://StringCounterTest/user/counter")
24   counterString ! "Hello World Again"
25   val counterInt =
26 system.actorFor[Int]("akka://StringCounterTest/user/counter")
27 // dynamic type error!
28   println("Hello") // will not be executed
29   counterInt ! 2    // will not be executed
30 }
31
32 /*
33 Terminal Output:
34
35 received 1 message(s):
36   Hello World
37 received 2 message(s):
38   Hello World Again
39 Exception in thread "main" java.lang.Exception:
40 ActorRef[akka://StringCounterTest/user/counter] does not exist
41 or does not have type ActorRef[Int]
42  */
```

Figure 3.1: TAkka Example: A String Counter

on the logic of the services rather than worrying about incoming messages of unexpected types.

3.2 Type-parameterized Actor

A TAkka actor has type `Actor[M]`. It inherits the Akka `Actor` trait to minimize implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor API. 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 type constructor `Actor` is *invariant* because the same type parameter is used for `ActorContext`, which is *invariant* as will be explained in Section 3.5. Finally, the actor context for the actor, `typedContext`, has type `ActorContext[M]`.

To maintain the actor behaviour and the supervision relationship, a special class of messages, which has type `SystemMessage`, should be handled by all actors. Unlike the Akka design, which handles system messages in the `receive` block, system messages are handled in a separate method, namely the `systemMessageHandler` method. System messages and its handler will be discussed in detail at Section 3.11.

```
1 package takka.actor
2
3 abstract class Actor[M:Manifest] extends akka.actor.Actor
4   protected def typedReceive:Function[M, Unit]
5
6   val typedSelf:ActorRef[M]
7   val typedContext:ActorContext[M]
8   var supervisorStrategy: SupervisorStrategy
9   def systemMessageHandler:SystemMessage => Unit
10
11  def preStart(): Unit
12  def preRestart(reason: Throwable, message: Option[Any]): Unit
13  def postRestart(reason: Throwable): Unit
14  def postStop(): Unit
```

Figure 3.2: TAkka API: Actor

Notice that the type of `typedReceive` is `Function[M, Unit]`, whereas the

type of `receive` in the Akka `Actor` class is `PartialFunction[Any, Unit]`. An advantage of using `Function` is that the compiler can check the completeness of the domain patterns. In Akka, 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 `Actor`, `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.

Types of variables and methods that do not related to message passing are not type parameterized (i.e. `supervisorStrategy`, `preStart`, `postRestart`, and `postStop`). The `preRestart` method has the same type as the Akka version. It is invoked before the actor is restarted due to an error (i.e. `reason`) that *might be* caused when processing a message (i.e. `message`). Because an actor can be evolve to a version that handles more types of messages (Section 3.6), the type of the problematic message cannot be determined in advance. It is fine to define some message patterns that will never be triggered at run-time.

Notice that `takka.actor.Actor` inherits `akka.actor.Actor`. A critical problem of using inheritance is that dynamically typed Akka API, which we are trying to avoid whenever possible, are still available to TAkka users. Unfortunately, this limitation cannot be overcome by using delegation because, as we have seen in the Akka API, a child actor is created by calling the `actorOf` method from its supervisor's actor context, which cannot be accessed outside the supervisor. `Actor` 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. To overcome the limitation, a complete reimplementation of that TAkka Actor library is required. The author estimates that the required work is similar to implementing the Akka Actor library.

3.3 Type-parameterized Actor Reference

The last section explains the type-parameterised Actor class, `Actor[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 unde-

fined 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 aimed at preventing ill-typed messages at the sender side. We achieve this 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.

```
1 package takka.actor
2
3 abstract class ActorRef[-M](implicit mt:Manifest[M])
4   val untypedRef:akka.actor.ActorRef
5
6   def !(message: M):Unit
7   def publishAs[SubM<:M](implicit smt:Manifest[SubM]):ActorRef[SubM]
8
9   abstract def path: akka.actor.ActorPath
10  final def compareTo(other: ActorRef[_]): Int
11  final def equals(that: Any): Boolean
12  // no forward method
```

Figure 3.3: TAkka API: Actor Reference

As 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 can usually 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

```
1 package sample.takka
2
3 import takka.actor.ActorRef
4 import takka.actor.ActorSystem
5 import takka.actor.Props
6 import takka.actor.Actor
7
8 sealed trait Operation
9 case class Multiplication(m:Int, n:Int) extends Operation
10 case class Division(m:Int, n:Int) extends Operation
11
12 class Calculator extends Actor[Operation] {
13   def typedReceive = {
14     case Multiplication(m:Int, n:Int) =>
15       println(m +" * "+ n +" = "+ (m*n))
16     case Division(m, n) =>
17       println(m +" / "+ n +" = "+ (m/n))
18   }
19 }
20
21 object TAkkaActorRefTest extends App{
22   val system = ActorSystem("MySystem")
23   val calculator:ActorRef[Operation] = system.actorOf(Props[Operation,
       Calculator], "calculator")
24
25   // explicit type conversion
26   val multiple = calculator.publishAs[Multiplication]
27
28   // implicit type conversion
29   val divide:ActorRef[Division] = calculator
30
31   calculator ! Multiplication(3, 2)
32   multiple ! Multiplication(3, 3)
33   // multiple ! Division(6, 2)
34   //Compiler Error: type mismatch; found : sample.takka.Division
35   //   required: sample.takka.Multiplication
36   divide ! Division(6, 2)
37 }
38
39 /*
40 Terminal Output:
41 3 * 2 = 6
42 3 * 3 = 9
43 6 / 2 = 3
44 */
```

Figure 3.4: TAkka Example: Typed Actor Reference

understand why `ActorRef` is contravariant, let us consider both the *Get and Put Principle* and an illustrative example. `ActorRef` is contravariant because users only `put` values to an actor reference (using the `!` method) but never get values out of it. Fetching the value sent via an actor reference is done in the `typedReceive` method of the `Actor` class. 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.

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. For example, line 26 in Figure 3.4 uses the `publishAs` method to explicitly cast an actor reference of type `ActorRef[Operation]` to its supertype `ActorRef[Multiplication]`. The author believes 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 supertypes and modify affected classes in the type hierarchy, some of which may not be accessible by application developers.

3.4 Type Parameterized Props

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

```
1 package takka.actor
2
3 final case class Props[-T](props:  akka.actor.Props)
4
5 object Props
6   def apply[T, A<:Actor[T]] (implicit arg0:Manifest[A]): Props[T]
7   def apply[T](clazz:  Class[_ <:  Actor[T]],args:Any*):  Props[T]
```

Figure 3.5: TAkka API: Props

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

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

In the above code, Scala checks that `StringCounter` is a subtype of `Actor[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 the API 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 API, TAkka methods take an additional type parameter: `M`, `SupM`, or `Msg`. The type variable `M` is the same as the type parameter of the `ActorContext` class, which corresponds to the type parameter of the `Actor` class. The later in turn is the same as the input type of its `typedReceive` method. Therefore, the `props` method has type `Props[M]` because it is used to create an actor of type `Actor[M]`. The `typedSelf` value has type `ActorRef[M]` because it records the actor reference to its corresponding actor. The `SupM` type variable in the `become` method performs a static check for backward compatible behaviour upgrade, a topic elaborated in Section 3.6. The `Msg` type variable in `actorOf`, `remoteActorOf`, and `actorFor` denotes the type parameter of another actor. Therefore, `Msg` has no relationship with the other two type variables. Notice that the type parameter `M` is used in a *covariant*

position in the `become` method and in a *contravariant* position is the `Props` type and the `ActorRef` type. To be valid in all cases, `ActorContext` is a *invariant* type constructor.

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. Notice that, 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 than calling `actorOf` because the IP address and port number need to be fetched from the system configuration. The way to enable distributed programming will be explained in Section 3.12.

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 resides. TAkka removes the API call that returns 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. The author recommends 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 API.

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

```
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

are handled by the `receive` method. In TAkka, the `ReceiveTimeout` message is handled by the `systemMessageHandler` method separately. Section 3.11 explains the TAkka design in depth.

Finally, TAkka defines the `become` method with a new signature so that behaviour upgrade is backward compatible. To guarantee backward compatibility, the `unbecome` method is removed. The next section explains behaviour upgrade in TAkka.

3.6 Backward Compatible Behaviour Upgrade

The `become` method enables behaviour upgrade 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 is separated from the handler for other messages, TAkka users may update the system message handler as well. Secondly, behaviour upgrade in TAkka *must* be backward compatible and *cannot* be rolled back. In other words, an actor must evolve to a version that is at least capable of handling all messages accepted by the previous version. The above decision was 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 `Actor[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 message handler is aware of all types of messages that may be sent to an actor before the upgrade. Backward compatible behaviour upgrade 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 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 records the manifest 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

```
1 package takka.actor
2
3 abstract class ActorContext[M:Manifest] {
4   implicit private var mt:Manifest[M] = manifest[M]
5
6   def become[SupM >: M](
7       behavior: SupM => Unit
8   )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] ={
9     become(behavior, this.systemMessageHandler,
          this.possiblyHarmfulHandler)
10  }
11
12  def become[SupM >: M](
13      behavior: SupM => Unit,
14      newSystemMessageHandler:SystemMessage=>Unit
15  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] = {
16    become(behavior, newSystemMessageHandler,
          this.possiblyHarmfulHandler)
17  }
18
19  def become[SupM >: M](
20      newTypedReceive: SupM => Unit,
21      newSystemMessageHandler:
22              SystemMessage => Unit
23      newpossiblyHarmfulHandler:akka.actor.PossiblyHarmful => Unit
24  )(implicit smtTag:Manifest[SupM]):ActorRef[SupM] = {
25     val smt = manifest[SupM]
26     if (!(smt >:> mt))
27       throw BehaviorUpdateException(smt, mt)
28
29    this.mt = smt
30    this.systemMessageHandler = newSystemMessageHandler
31    this.possiblyHarmfulHandler = newpossiblyHarmfulHandler
32
33    new ActorRef[SupM] {
34      val untypedRef = untypedContext.self
35    }
36  }
37 }
38
39 case class BehaviorUpdateException(smt:Manifest[_], mt:Manifest[_])
      extends Exception(smt + "must be a supertype of "+mt+".")
```

Figure 3.7: Behaviour Upgrade in TAkka

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 how to upgrade the message handler in TAkka. The code is similar to the Akka example in Figure 2.6. The calculator server begins with a basic version, which 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 upgrade 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 (Figure 2.6). First, two new types, namely `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 can 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]` must always ensure that a division request can be understood by the server.

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, TAkka provides a typed name server which maps each registered typed name to a value of the corresponding type, and allows look up of a value by giving a typed name. Our typed name server can be straightforwardly ported to other platforms that support type reflection.

```
1 package sample.takka
2 import takka.actor.{ActorRef, ActorSystem, Props, Actor}
3
4 trait Operation
5 trait BasicOperation extends Operation
6 case class Multiplication(m:Int, n:Int) extends BasicOperation
7 case class Upgrade[Op >:  BasicOperation](advancedCalculator:Op=>Unit)
8                                           extends BasicOperation
9 class CalculatorServer extends Actor[BasicOperation] {
10  def typedReceive = {
11    case Multiplication(m:Int, n:Int) =>
12      println(m +" * "+ n +" = "+ (m*n))
13    case Upgrade(advancedCalculator) =>
14      println("Upgrading ...")
15      typedContext.become(advancedCalculator)
16  }
17 }
18 object CalculatorUpgrade extends App {
19  val system = ActorSystem("CalculatorSystem")
20  val simpleCal:ActorRef[BasicOperation] =
        system.actorOf(Props[BasicOperation, CalculatorServer],
        "calculator")
21  simpleCal ! Multiplication(5, 1)
22
23  case class Division(m:Int, n:Int) extends Operation
24  def advancedCalculator:Operation=>Unit = {
25    case Multiplication(m:Int, n:Int) =>
26      println(m +" * "+ n +" = "+ (m*n))
27    case Division(m:Int, n:Int) =>
28      println(m +" / "+ n +" = "+ (m/n))
29    case Upgrade(_) =>
30      println("Upgraded.")
31  }
32  simpleCal ! Upgrade(advancedCalculator)
33  val advancedCal =
        system.actorFor[Operation]("akka://CalculatorSystem/user/calculator")
34  advancedCal ! Multiplication(5, 3)
35  advancedCal ! Division(10, 3)
36  advancedCal ! Upgrade(advancedCalculator)
37 }
38 /*  Terminal Output:
39 5 * 1 = 5
40 Upgrading ...
41 5 * 3 = 15
42 10 / 3 = 3
43 Upgraded.
44  */
```

Figure 3.8: TAkka Example: Backward Compatible Behaviour Upgrade

```
1 package takka.nameserver
2
3 import scala.collection.mutable.HashMap
4 import scala.reflect.Manifest
5 import scala.Symbol
6
7 case class TSymbol[-T:Manifest](val s:Symbol) {
8    private [takka] val t:Manifest[_] = manifest[T]
9    override def hashCode():Int = s.hashCode()
10 }
11
12 case class TValue[T:Manifest](val value:T){
13  private [takka] val t:Manifest[_] = manifest[T]
14 }
15
16 object NameServer {
17  private val nameMap = new HashMap[TSymbol[_], TValue[_]]
18
19  def set[T:Manifest](name:TSymbol[T], value:T):Boolean = synchronized {
20    val tValue = TValue[T](value)
21    if (nameMap.contains(name)){ return false }
22    else{
23      nameMap.+=((name, tValue))
24      return true
25    }
26  }
27
28  def unset[T](name:TSymbol[T]):Boolean = synchronized {
29    if (nameMap.contains(name) && nameMap(name).t <:< name.t ){
30      nameMap -= name
31      return true
32    }else{ return false }
33  }
34
35  def get[T](name:TSymbol[T]):Option[T] = synchronized {
36    if (!nameMap.contains(name)) {return None}
37    else {
38      val tValue = nameMap(name)
39      if (tValue.t <:< name.t) { return
            Some(tValue.value.asInstanceOf[T]) }
40      else{ return None }
41    }
42  }
43 }
```

Figure 3.9: Typed Name Server

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.

As will be explained in the next section, the typed name server permits the case where a value is fetched and used as a value of its supertype. For efficiency considerations, the `hashCode` method of `TSymbol` does not count the value of its type descriptor. In Scala, types of different fully qualified names are considered different. Hence, the value of a type descriptor only records its fully qualified name rather than the structure of a type.

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`

  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 API 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 Section 3.8 will explain, 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 fewer 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, the design prevents the situation where users accidentally register two typed names with the same symbolic representation but 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 API 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 for 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

The same as in 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 API 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 delegating some tasks to an Akka actor system, a TAkka actor system uses the typed name server (Section 3.7) 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 for enquiring 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. Unlike the Akka `actorFor` method, which always return an actor reference that may point to a dummy actor where all messages sink, the TAkka `actorFor` method either returns an actor reference pointing an alive actor or throws an Exception if no matched actor is found. The process is of fetching a TAkka actor reference summarised in Figure 3.13.

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

```
1 package takka.actor
2
3 case class NotRemoteSystemException(system:ActorSystem) extends
      Exception("ActorSystem: "+system+" does not support remoting")
4
5 object ActorSystem
6     def apply():ActorSystem
7     def apply(sysname: String): ActorSystem
8     def apply(sysname: String, config: Config): ActorSystem
9     def apply(sysname: String, config: Config,
10                classLoader: ClassLoader): ActorSystem
11
12 abstract class ActorSystem
13     private val system:akka.actor.ActorSystem
14     def stop(actorRef:ActorRef[_])
15     def deadLetters : ActorRef[Any]
16     def isTerminated: Boolean
17
18     def actorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
19     def actorOf[Msg:Manifest](props:Props[Msg],
20                               name:String):ActorRef[Msg]
21
22     @throws(classOf[NotRemoteSystemException])
23     def remoteActorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
24     @throws(classOf[NotRemoteSystemException])
25     def remoteActorOf[Msg:Manifest](props:Props[Msg],
26                                     name:String):ActorRef[Msg]
27
28     def actorFor[M:Manifest](actorPath: String): ActorRef[M]
29     def actorFor[M:Manifest](actorPath: akka.actor.ActorPath):
         ActorRef[M]
```

Figure 3.10: TAkka API: Actor System

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 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.

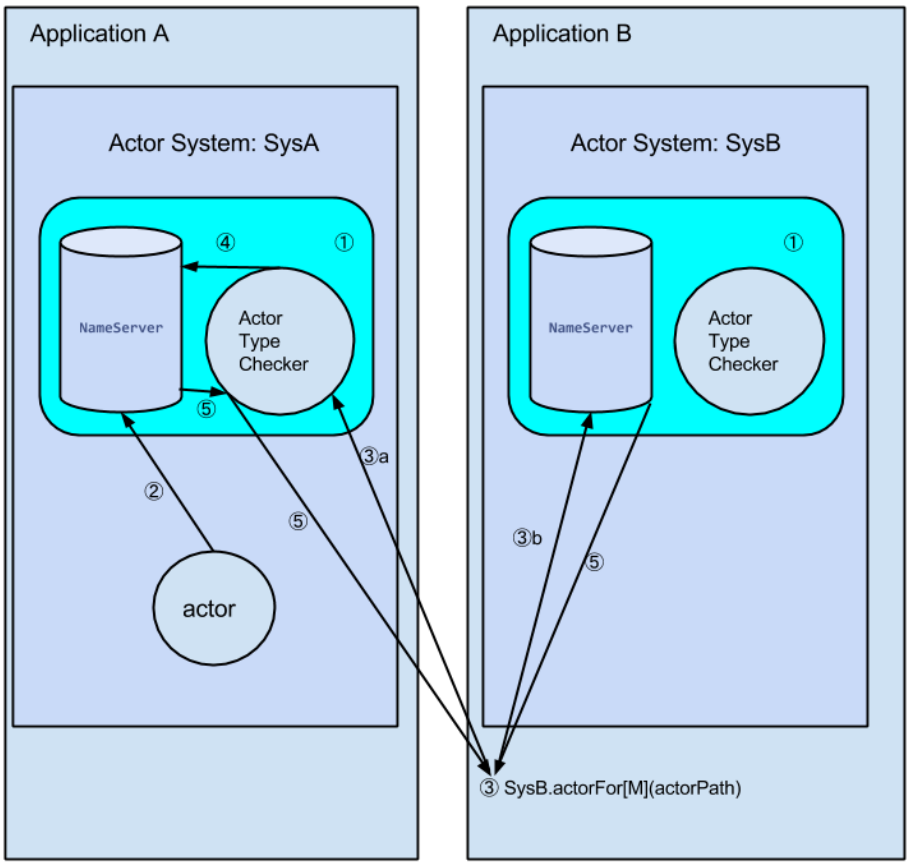
The `actorFor` method (Figure 3.12) returns a typed actor reference if there

```
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

```
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



① When an actor system is initialised, it creates a typed name server and an actor type checker. The actor type checker is an Akka actor.
② When an actor is created, its path and type is registered at the typed nameserver of the actor system where that actor is created.
③ When SysB.actorFor[M](actorPath) is called in an Scala application
③a If actorPath contains host name, port number, and acotor system name, a type checking message is sent to the actor type checker of that remote actor system (e.g. SysA).
③b Otherwise, try to fetch the typed actor reference from local typed name server.
④ The actor type checker tries to fetch the typed actor reference from typed name server.
⑤ Return Some(actorRef) if there is an actor path with a matched type; return None other wise.

Figure 3.13: Fetched Typed Actor Reference

is such with an expected type located at a given path. Figure 3.13 illustrates how typed name servers are used in TAkka to fetch a potentially remote actor reference. 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 `ActorTypeChecker` 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 considered in this project.

Figure 3.14 gives the API of supervisor strategies in TAkka. In fact, it is the same as the Akka version given in Figure 2.7. TAkka reuses the Akka API because none of the supervisor strategies requires type-parameters and TAkka separates the message handler for system messages and the message handler for other messages.

A TAkka Safe Calculator example is given in Figure 3.15 as a reminder of the Akka Safe Calculator in Figure 2.8. Both examples define a simple calculator which supports multiplication and division. The simple calculator does

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

sealed trait Directive extends AnyRef
object Escalate extends Directive
object Restart extends Directive
object Resume extends Directive
object Stop extends Directive
```

Figure 3.14: TAkka API: Supervisor Strategies

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 it 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. The supervisor strategy used in the TAkka implementation is exactly the same as the one in the Akka implementation.

```
1 package sample.takka
2 import takka.actor.{ActorRef, ActorSystem, Props, Actor}
3 sealed trait Operation
4 case class Multiplication(m:Int, n:Int) extends Operation
5 case class Division(m:Int, n:Int) extends Operation
6 class Calculator extends Actor[Operation] {
7   def typedReceive = {
8     case Multiplication(m:Int, n:Int) =>
9       println(m +" * "+ n +" = "+ (m*n))
10    case Division(m, n) =>
11      println(m +" / "+ n +" = "+ (m/n))
12  }
13 }
14 class SafeCalculator extends Actor[Operation] {
15   import language.postfixOps
16   override val supervisorStrategy =
17     OneForOneStrategy(maxNrOfRetries = 2, withinTimeRange = 1 minute) {
18       case _: ArithmeticException =>
19         println("ArithmeticException Raised to: "+self)
20         Restart
21     }
22   val child:ActorRef[Operation] =
23         typedContext.actorOf(Props[Operation, Calculator], "child")
24   def typedReceive = {   case m => child ! m }
25 }
26 object SupervisorTest extends App{
27   val system = ActorSystem("MySystem")
28   val calculator:ActorRef[Operation] =
29        system.actorOf(Props[Operation, Calculator], "calculator")
30   val multiplicator = calculator.publishAs[Multiplication]
31
32   calculator ! Multiplication(3, 2)
33   multiplicator ! Multiplication(3, 3)
34 // multiplicator ! Division(6, 2)
35 // Compiler Error: type mismatch; found : sample.takka.Division
36 // required:  sample.takka.Multiplication
37   calculator ! Division(10, 0)
38   calculator ! Division(10, 5)
39 }
40 /* Terminal Output:
41 3 * 2 = 6
42 3 * 3 = 9
43 java.lang.ArithmeticException: / by zero
44 ArithmeticException Raised to:
     Actor[akka://MySystem/user/safecalculator]
45 10 / 5 = 2
46 */
```

Figure 3.15: TAkka Example: Supervised Calculator

3.10 Fixing The Type Pollution Problem

In a system with multiple components, different components may require different interfaces; since all messages are received in the same mailbox, a naive approach would be to set the type to the union of all the interfaces, causing each component to see a type containing messages not intended for it—an issue dubbed the Type Pollution Problem.

This section illustrates the Type Pollution Problem and its solution on an instance of the Model-View-Controller pattern [Reenskaug, 1979; Burbeck, 1987]. The Model and View have separate interfaces to the Controller, and neither should see the interface used by the other. However, the naive approach would have the Controller message type contain all the messages the Controller receives, from both the Model and the View. A similar problem can occur in a multi-tier architecture [Fowler, 2002], where an intermediary layer interfaces with both the layer above and the layer below.

One solution to the type pollution problem is using separate channels for distinct parties. For instance, in Model-View-Controller, one channel would communicate between Model and Controller, and a distinct channel communicate between Model and View. Programming models that support this solution includes the join-calculus [Fournet and Gonthier, 2000] and the typed π-calculus [Sangiorgi and Walker, 2001]. Can we gain similar advantages for a system based on actors rather than channels?

The TAkka solution is to publish a component at different types to different parties. The published type must be a supertype of the most precise type of the component. In Java and Scala applications, this solution can be tricky because, as will be briefly discussed in Section 3.10.3, 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 a graphical user interface (GUI) implemented using the MVC pattern.

3.10.1 Case Study: Tic-Tac-Toe

3.10.1.1 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 or she succeeds in placing three respective marks, i.e. three Xs or three Os,

in a horizontal, vertical, or diagonal row. The game is drawn if no player wins when the grid is fully marked.

Figure 3.16 gives an example game won by the first player, X. Figures 3.16a to 3.16h 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 with the next move. The middle of the window shows the current status of the game board. The right hand side contains control buttons, each of which kills one component of the application and test if that component will be restarted. Finally, Figure 3.16i announces the winner.

3.10.1.2 The MVC pattern

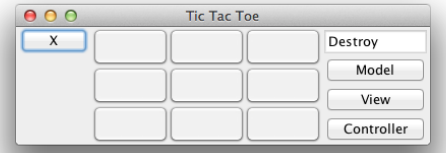
Model-view-controller (MVC) is a software architecture pattern introduced by Reenskaug [1979]. The pattern separates the data abstraction (the model), the representation of data (the view), and the component for manipulating the data and interpreting user inputs (the controller). In an application built using MVC, a controller coordinates a model and a view, for example, sending instructions and reacting to messages from the model and the view. Consequently, the controller has two distinct sets of interface: one to work with the model, the other to work with 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., 2008], to mobile applications including Apple iOS applications [Apple Inc., 2012]. This section will follow the MVC pattern to implement a Tic-Tac-Toe Game with a GUI. The challenge here is using types to prevent the situation where the model sends the controller a message expected from the view, or the view pretends to be the model.

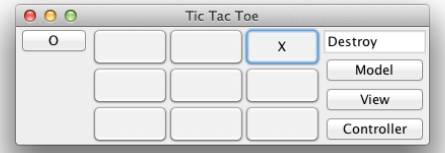
3.10.2 A TAkka Implementation

TAkka solves the type pollution problem by using subtyping polymorphism. The code from Figure 3.17 to Figure 3.20 gives an TAkka application that implements the Tic-Tac-Toe game with a GUI. The code marked in blue may be reused by other applications built using the MVC pattern.

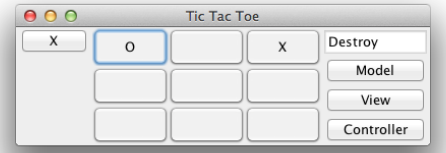
Messages used in this implementation are given in Figure 3.17. Messages sent to the controller are separated into two groups: those expected from the model and those expected from the view. The `Controller` actor of this application, defined in Figure 3.20, reacts to messages sent either from the `Model` actor



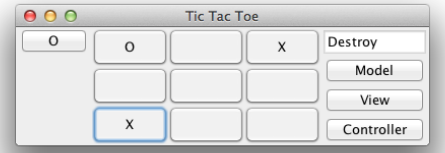
(a)



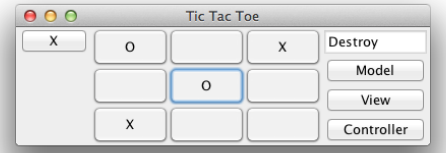
(b)



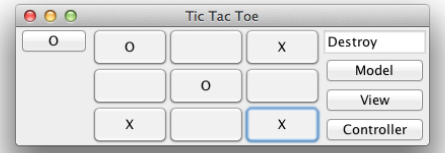
(c)



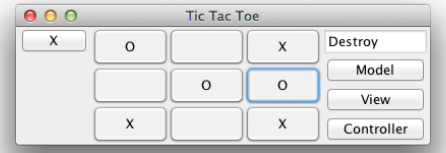
(d)



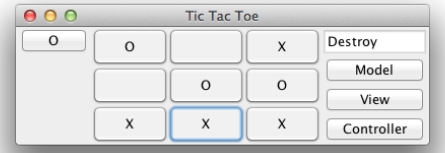
(e)



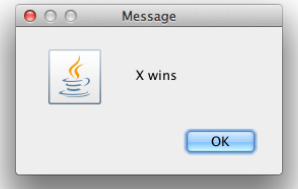
(f)



(g)



(h)



(i)

Figure 3.16: A Game of Tic-Tac-Toe

```
1 package sample.tic_tac_toe.takka
2
3 sealed trait ControllerMessage
4 sealed trait View2ControllerMessage extends ControllerMessage
5 final case class ButtonClickedAt(row:Int, col:Int) extends
      View2ControllerMessage
6
7 sealed trait Model2ControllerMessage extends ControllerMessage
8 final case class GridNotEmpty(row:Int, col:Int) extends
      Model2ControllerMessage
9 final case class PlayedCross(row:Int, col:Int) extends
      Model2ControllerMessage
10 final case class PlayedO(row:Int, col:Int) extends
      Model2ControllerMessage
11 final case class NextMove(move:Move) extends Model2ControllerMessage
12 final case class Winner(move:Move) extends Model2ControllerMessage
13
14 sealed trait Controller2ViewMessage
15 final case class DisplyError(err:String) extends Controller2ViewMessage
16 final case class DrawCross(row:Int, col:Int) extends
      Controller2ViewMessage
17 final case class DrawO(row:Int, col:Int) extends Controller2ViewMessage
18 final case class DisplayNextMove(move:Move) extends
      Controller2ViewMessage
19 final case class AnnounceWinner(winner:Move) extends
      Controller2ViewMessage
20
21 sealed trait Controller2ModelMessage
22 final case class MoveAt(row:Int, col:Int) extends
      Controller2ModelMessage
23
24 final case class
      ModelSetController(controller:ActorRef[Model2ControllerMessage])
      extends Controller2ModelMessage
25 final case class
      ViewSetController(controller:ActorRef[View2ControllerMessage])
      extends Controller2ViewMessage
26
27
28 sealed trait Move
29 final case object X extends Move
30 final case object O extends Move
```

Figure 3.17: TicTacToe: Message

```
1 package sample.tic_tac_toe.takka
2 import takka.actor._
3 final class Model extends Actor[Controller2ModelMessage] {
4  var controller:ActorRef[Model2ControllerMessage] = _
5  def typedReceive = {
6    case ModelSetController(control) => controller = control
7    case MoveAt(row:Int, col:Int) => { model.setStatus(row, col) }
8  }
9  private object model {
10   sealed trait GridStatus
11   case object Empty extends GridStatus
12   case object XModelMove extends GridStatus
13   case object OModelMove extends GridStatus // Uppercase O
14
15   var nextXMove:Boolean = true // true->X false->O
16   val status:Array[Array[GridStatus]] =
17        Array(Array(Empty, Empty, Empty),
18              Array(Empty, Empty, Empty),
19              Array(Empty, Empty, Empty))
20   def setStatus(row:Int, col:Int) = {
21     if(nextXMove){
22       if (status(row)(col) == Empty) {
23         status(row)(col) = XModelMove
24         controller ! PlayedCross(row, col)
25         nextXMove = false
26         controller ! NextMove(O)
27       }else{ controller ! GridNotEmpty(row, col)    }
28     }else{
29       if (status(row)(col) == Empty) {
30         status(row)(col) = OModelMove
31         controller ! PlayedO(row, col)
32         nextXMove = true
33         controller ! NextMove(X)
34       }else{ controller ! GridNotEmpty(row, col)    }
35     }
36
37     checkWinner match {
38       case Empty =>
39       case XModelMove =>      controller ! Winner(X)
40       case OModelMove =>      controller ! Winner(O)
41    }}
42  def checkWinner:GridStatus = {
43    // reuse GridStatus instead of a new set of values
44    // return XModelMove if X wins
45    // return OModelMove if O wins
46    // return Empty if no winner has
47 }}
```

Figure 3.18: 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 Actor[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 3.19: TicTacToe: View

```
1 package sample.tik_tak_tok.takka
2
3 import takka.actor._
4
5 final class Controller(model:ActorRef[Controller2ModelMessage],
    view:ActorRef[Controller2ViewMessage]) extends
    Actor[ControllerMessage] {
6   def typedReceive = {
7     case ButtonClickedAt(row, col) =>
8       model ! MoveAt(row, col)
9     case GridNotEmpty(row, col) =>
10      view ! DisplyError("grid "+row+" , "+col+" is not empty")
11    case PlayedCross(row, col) =>
12      view ! DrawCross(row, col)
13    case PlayedO(row, col) =>
14      view ! DrawO(row:Int, col:Int)
15    case NextMove(move) =>
16      view ! DisplayNextMove(move)
17    case Winner(move) =>
18      view ! AnnounceWinner(move)
19  }
20
21  override def preStart() = {
22    model !
        ModelSetController(typedSelf.publishAs[Model2ControllerMessage])
23    view !
        ViewSetController(typedSelf.publishAs[View2ControllerMessage])
24  }
25 }
26
27 package sample.tic_tac_toe.takka
28
29 import takka.actor._
30 object TicTacToeApplication extends App {
31   val system = ActorSystem("LocalTicTacToe")
32   val model = system.actorOf(Props[Controller2ModelMessage, Model],
       "model")
33   val view = system.actorOf(Props[Controller2ViewMessage, View], "view")
34   val controller = system.actorOf(Props(new Controller(model, view)),
       "controller")
35 }
```

Figure 3.20: TicTacToe: Application

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 3.20 can be committed because the type of the controller has been refined in the `ModelSetController` message and the `ViewSetController` message, the code explicitly expresses the type convention and lets the compiler double check the type.

In the definition of the `Model` actor (Figure 3.18) and the `View` actor (Figure 3.19), 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 dimensional 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 is 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. Detailed GUI implementation is omitted in this thesis for clarity. Readers can found the complete code in the public code repository of this project. [HE, 2014a]

Finally, setting up the application is straightforward. In the code given at the bottom of Figure 3.20, 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 3.16.

3.10.3 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 3.21 gives an interface of implementing the Tic-Tac-Toe game using the MVC pattern without actors. The code marked in blue 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 3.21 separates the methods of a controller to be called by a model and those to be called by a view into 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 shall be added to either of the two traits, the whole program needs to be recompiled and re-deployed. Where an application is a collaborative project maintained by different groups, attempts at large-scale updates should be avoided whenever possible.

In contrast, our TAkka implementation avoids the problems suffered by the simple Scala solution because new messages can be added easily as a subtype of an earlier defined message. With the benefit of backward compatible behaviour upgrading (Section 3.6), the controller, the model and the view can be updated separately.

3.11 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 defines messages of the same name as subclasses of the `SystemMessage` trait.

Message	TAkka 2.0	TAkka 2.1	Akka 2.0	Akka 2.1
Kill	public	public	public	public
PoisonPill	public	public	public	public
ReceiveTimeout	public	public	public	public
ChildTerminated	public	private	public	private
Restart	public	private	public	private
Terminated	not included	not included	public	public
Create	private	private	public	private
Failed	private	private	public	private
Link	not included	not included	public	private
Unlink	not included	not included	public	private
Suspend	private	private	public	private
Resume	private	private	public	private

Table 3.1: System Messages

Table 3.1 lists system messages used in Akka and TAkka. Some messages are defined as public objects which can be handled by a user where as some

```
1 package sample.tic_tac_toe.mvcobject
2
3 trait Controller extends ControllerForView with ControllerForModel
4 class GameController(model:Model, view:View) extends Controller {
5   // implementation
6 }
7
8 trait ControllerForView {
9   def buttonClickedAt(row:Int, col:Int):Unit
10 }
11 trait ControllerForModel {
12   def gridNotEmpty(row:Int, col:Int):Unit
13   def playedCross(row:Int, col:Int):Unit
14   def playedO(row:Int, col:Int):Unit
15   def nextMove(move:Move):Unit
16   def winner(move:Move):Unit
17 }
18
19 trait Model {
20   def setController(controller:ControllerForModel):  Unit
21   def moveAt(row:Int, col:Int): Unit
22 }
23 class GameModel extends Model {
24   // implementation
25 }
26
27 trait View {
28   def setController(controller:ControllerForView):  Unit
29   def displyError(err:String): Unit
30   def drawCross(row:Int, col:Int): Unit
31   def drawO(row:Int, col:Int): Unit
32   def displayNextMove(move:Move): Unit
33   def announceWinner(winner:Move): Unit
34 }
35 class GameView extends View {
36   // implementation
37 }
38
39 sealed trait Move
40 final case object X extends Move
41 final case object O extends Move
42
43 object TicTacToeApplication extends App {
44   val model = new GameModel;
45   val view = new GameView;
46   val controller = new GameController(model, view)
47 }
```

Figure 3.21: TicTacToe: MVC Interface

messages are defined as private objects which only be used by the library developers. TAkka retains `Kill` and `PoisonPill` because it is used in the Chaos Monkey library and the Supervision View library explained in Section 5.4. TAkka users may also want to use those two messages for reliability testing. The `ReceiveTimeout` message is retained because it is used in many applications considered in this thesis. The TAkka library makes `Create`, `Failed`, `Suspend`, and `Resume` private because reactions to those messages can be consistently defined in library rather than leave to users. The decision is verified by the design of Akka 2.1, which makes all above messages private as well. Akka 2.1 also makes `ChildTerminated` and `Restart` private for the same reason. The author does not recognise this point in the design of TAkka 2.0 and follow the Akka design in TAkka 2.1. The TAkka library does not retain `Terminated`, `Link`, and `Unlink` because, as will be explained in Section 3.13, life-cycle monitor relationship outside the supervision tree is considered as a redundant design.

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 sends itself a `Failed(cause: Throwable)` message when an error of `cause` occurs 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 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, the `SystemMessage` trait is defined 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.

3.12 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.

3.12.1 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 directed to look at the Akka documentation for the version they are using.

The configuration in Figure 3.22 is used for the `RemoteCreation` system in Figure 3.25. The configuration overrides three system policies. Firstly, the system enables 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 3.22: Configuration Example: Distributed Creation

3.12.2 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 3.24 and the `AdvancedCalculatorActor` in Figure 3.23. 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`

```
1 package typed.remote.calculator
2 import takka.actor.{Actor, ActorRef}
3 import akka.actor.ActorPath
4 import scala.reflect.runtime.universe._
5
6 trait MathOp
7 case class Add(nbr1: Int, nbr2: Int) extends MathOp
8 case class Subtract(nbr1: Int, nbr2: Int) extends MathOp
9 case class Multiply(nbr1: Int, nbr2: Int) extends MathOp
10 case class Divide(nbr1: Double, nbr2: Int) extends MathOp
11
12 trait CalculatorMessage
13 case class Op(op:MathOp, sender:ActorRef[MathResult])
14     extends CalculatorMessage
15
16 trait MathResult
17 case class AddResult(nbr: Int, nbr2: Int, result: Int)
18     extends MathResult
19 case class SubtractResult(nbr1: Int, nbr2: Int, result: Int)
20     extends MathResult
21 case class MultiplicationResult(nbr1: Int, nbr2: Int, result: Int)
22     extends MathResult
23 case class DivisionResult(nbr1: Double, nbr2: Int, result: Double)
24     extends MathResult
25 case class Ask(calculator:ActorRef[CalculatorMessage], op:MathOp)
26     extends MathResult
27
28
29
30 class AdvancedCalculatorActor extends Actor[CalculatorMessage] {
31   def typedReceive = {
32     case Op(Multiply(n1, n2), sender) =>
33       println("Calculating %d * %d".format(n1, n2))
34       sender ! MultiplicationResult(n1, n2, n1 * n2)
35     case Op(Divide(n1, n2), sender) =>
36       println("Calculating %.0f / %d".format(n1, n2))
37       sender ! DivisionResult(n1, n2, n1 / n2)
38   }
39 }
```

Figure 3.23: Distributed Calculator: Messages and Advanced Calculator

```
1 package typed.remote.calculator
2 import akka.kernel.Bootable
3 import takka.actor.{ Props, Actor, ActorSystem }
4 import com.typesafe.config.ConfigFactory
5
6 class SimpleCalculatorActor extends Actor[CalculatorMessage] {
7   def typedReceive = {
8     case Op(Add(n1, n2), sender) =>
9       println("Calculating %d + %d".format(n1, n2))
10      sender ! AddResult(n1, n2, n1 + n2)
11    case Op(Subtract(n1, n2), sender) =>
12      println("Calculating %d - %d".format(n1, n2))
13      sender ! SubtractResult(n1, n2, n1 - n2)
14  }
15 }
16 class CalculatorApplication extends Bootable {
17   val system = ActorSystem("CalculatorApplication",
18               ConfigFactory.parseString("""
19                 akka{
20                   actor {
21                     provider = "akka.remote.RemoteActorRefProvider"
22                   }
23                   remote {
24                     netty {
25                       hostname = "129.215.91.88"
26                       port = 2552
27                     }
28                   }
29                 }""") )
30   val cal = system.actorOf(Props[CalculatorMessage,
31                           SimpleCalculatorActor], "simpleCalculator")
32 }
33
34 object CalcApp {
35   def main(args: Array[String]) {
36     new CalculatorApplication
37     println("Started Calculator Application - waiting for messages")
38   }
39 }
```

Figure 3.24: Distributed Calculator: Simple Calculator Local Creation

```
1 package typed.remote.calculator
2 import akka.kernel.Bootable
3 import com.typesafe.config.ConfigFactory
4 import scala.util.Random
5 import takka.actor._
6 class CreationApplication extends Bootable {
7   val system = ActorSystem("RemoteCreation",
8                            ConfigFactory.parseString("""
9                 akka{
10                  actor {
11                    provider = "akka.remote.RemoteActorRefProvider"
12                    deployment {
13                      /advancedCalculator {
14                        remote =
15                              "akka://CalculatorApplication@129.215.91.88:2552"
16                  }}}
17                  remote {
18                    netty {
19                      hostname = "129.215.91.195"
20                      port = 2554
21                }}}""") )
22   val localActor = system.actorOf(Props[MathResult, CreationActor],
23                                                   "creationActor")
24   val remoteActor = system.actorOf(Props[CalculatorMessage,
25                          AdvancedCalculatorActor], "advancedCalculator")
26   def doSomething(op: MathOp) = { localActor ! Ask(remoteActor, op) }
27 }
28 class CreationActor extends Actor[MathResult] {
29   def typedReceive = {
30     case Ask(calculator, op) =>
31       calculator ! Op(op, typedRemoteSelf)
32     case result: MathResult => result match {
33       case MultiplicationResult(n1, n2, r) =>
34         println("Mul result: %d * %d = %d".format(n1, n2, r))
35       case DivisionResult(n1, n2, r) =>
36         println("Div result: %.0f / %d = %.2f".format(n1, n2, r))
37 }}}
38 object CreationApp extends App {
39     val app = new CreationApplication
40     while (true) {
41       if (Random.nextInt(100) % 2 == 0)
42          app.doSomething(Multiply(Random.nextInt(20),
43                                   Random.nextInt(20)))
44       else app.doSomething(Divide(Random.nextInt(10000),
45                                   (Random.nextInt(99) + 1)))
46       Thread.sleep(200)
47 }}
```

Figure 3.25: Distributed Calculator: Distributed Creation

```
package typed.remote.calculator
import akka.kernel.Bootable
import scala.util.Random
import com.typesafe.config.ConfigFactory
import takka.actor.{ ActorRef, Props, Actor, 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 Actor[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 3.26: Distributed Calculator: Actor Look up

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.

3.13 Design Alternatives

Akka Typed Actor In the Akka library, there is a special class called `TypedActor`. Although an instance of `TypedActor` can be supervised by a standard actor, it is essentially a different framework as a service of `TypedActor` class is invoked by a method invocation instead of sending a message. Code in Figure 3.27 shows how to define a simple string processor using Akka typed actor. Line 34 to 36 show that a `TypedActor` object does not have a `!` method. Line 37 to 39 and their output (Line 44 to 45) show that an actor is located at the given address but messages sent to that actor using its actor reference are unhandled.

The Akka `TypedActor` class prevents some type errors but have two limitations. Firstly, `TypedActor` does not permit behaviour upgrade. Secondly, avoiding the type pollution problem, explained in Section 3.10, by using Akka typed actors is the same inconvenience as using a simple 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. It is recommended that the state of an actor, if there is any, to be saved in an *ETS* table, a data structure provided by the OTP library [Ericsson AB., 2013b]. In Scala, users are free to use mutable variables in code. 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

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

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 that 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 expand a great deal of 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.

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. The author believes 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.

3.14 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 look 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 other messages, supervision tree and remote communication can be done in the same way as in the Akka library.

Chapter 4

Evolution, Not Revolution

A condensed version of the material in this chapter appears
in [He et al., 2014, Section 5]

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 an equivalent generic version (evolution), rather than requiring generic types everywhere (revolution) [Naftalin and Wadler, 2006].

Figure 2.8 and Figure 3.15 presents how to define and use a safe calculator in the Akka and TAkka systems respectively. Think of a `SafeCalculator` actor as a service and its reference as a client interface. The following sections show how to upgrade the Akka version to the TAkka version gradually, either upgrading the service implementation first or the client interface.

4.1 TAkka Service with Akka Client

It is often the case that an actor-based service is implemented by one organization but used in a client application implemented by another. Let us assume that a developer decides to upgrade the service using TAkka actors, for example, by upgrading the Socko Web Server [Imtarnasan and Bolton, 2012], the Gatling stress testing tool [Excilys Group, 2012], or the core library of Play [Typesafe Inc. (c), 2013], as we do in Section 5.1. Will the upgrade affect legacy client applications built using the Akka library? Fortunately, no changes are required at all.

As the TAkka Actor class inherits the Akka `Actor` class, it can be used to create an Akka actor. For example, the object `akkaCal`, created at line 5 in Figure 4.1, is created from a TAkka actor and used as an Akka actor reference. After the service developer has upgraded all actors to equivalent TAkka versions, the developer may want to start a TAkka actor system. Until that time, the developer can create TAkka actor references but publish their untyped version to users who are working in the Akka environment (line 19). As a result, no changes are required for a client application that uses Akka actor references. Because an Akka actor reference accepts messages of any type, messages of unexpected type may be sent to TAkka actors. As a result, handlers for the `UnhandledMessage` event is required in a careful design (line 10 and 20).

```
1 import sample.takka.SafeCalculator.SafeCalculator
2
3 object TSAC extends App {
4   val akkasystem = akka.actor.ActorSystem("AkkaSystem")
5   val akkaCal = akkasystem.actorOf(
6     akka.actor.Props[SafeCalculator], "acal")
7   val handler = akkasystem.actorOf(
8     akka.actor.Props(new MessageHandler(akkasystem)))
9
10  akkasystem.eventStream.subscribe(handler,
11     classOf[UnhandledMessage]);
12  akkaCal ! Multiplication(3, 1)
13  akkaCal !  "Hello Akka"
14
15  val takkasystem = takka.actor.ActorSystem("TAkkaSystem")
16  val takkaCal = takkasystem.actorOf(
17     takka.actor.Props[String, TAkkaStringActor], "tcal")
18
19  val untypedCal= takkaCal.untypedRef
20  takkasystem.system.eventStream.subscribe(
21    handler,classOf[UnhandledMessage]);
22  untypedCal ! Multiplication(3, 2)
23  untypedCal ! "Hello TAkka"
24 }
25 /* Terminal output:
26 3 * 1 = 3
27 unhandled message:Hello Akka
28 3 * 2 = 6
29 unhandled message:Hello TAkka
30 */
```

Figure 4.1: TAkka Service with Akka Client

4.2 Akka Service with TAkka Client

Sometimes developers want to update the client code or API before upgrading the service implementation. For example, a developer may not have access to the service implementation; or the service implementation 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 calculator, initialise it in an Akka actor system, and obtain an Akka actor reference. Then, we wrap the Akka actor reference as a TAkka actor reference, `takkaCal`, which only accepts messages of type `Operation`.

```
import sample.takka.SafeCalculator.SafeCalculator

object TSAC extends App {
  val akkasystem = akka.actor.ActorSystem("AkkaSystem")
  val akkaCal = akkasystem.actorOf(
    akka.actor.Props[SafeCalculator], "acal")
  val handler = akkasystem.actorOf(
    akka.actor.Props(new MessageHandler(akkasystem)))

  akkasystem.eventStream.subscribe(handler,
     classOf[UnhandledMessage]);
  akkaCal ! Multiplication(3, 1)
  akkaCal !  "Hello Akka"

  val takkasystem = takka.actor.ActorSystem("TAkkaSystem")
  val takkaCal = takkasystem.actorOf(
     takka.actor.Props[String, TAkkaStringActor], "tcal")

  val untypedCal= takkaCal.untypedRef
  takkasystem.system.eventStream.subscribe(
    handler,classOf[UnhandledMessage]);
  untypedCal ! Multiplication(3, 2)
  untypedCal ! "Hello TAkka"
}
/* Terminal output:
3 * 1 = 3
unhandled message:Hello Akka
3 * 2 = 6
unhandled message:Hello TAkka
*/
```

Figure 4.2: TAkka Service with Akka Client

Chapter 5

TAkka: Evaluation

A condensed version of the material in this chapter appears
in [He et al., 2014, Section 6, 7 and 8]

This chapter evaluates the TAkka library with regards to the following three aspects. Sections 5.1 to 5.3 show that rewriting Akka programs using TAkka will *not* bring obvious code-size and runtime overheads. Moreover, Section 5.4 gives two accessory libraries, ChaosMonkey and SupervisionView, for testing the reliability and availability of TAkka applications.

5.1 Expressiveness

To assess the expressiveness of the TAkka library. The author selected examples from Erlang QuviQ [Arts et al., 2006] and open source Akka projects to ensure that the main requirements for actor programming were not unintentionally neglected. Section 5.1.1 lists examples ported from other projects. Examples from Erlang QuviQ were re-implemented using both Akka and TAkka. Examples from Akka projects were re-implemented using TAkka. Section 5.1.2 gives the evaluation results in term of code size and and type error detection.

5.1.1 Examples

5.1.1.1 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. QuviQ is a commercial product. The author gratefully acknowledges Thomas Arts from QuviQ.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. For copyright reason, code for this example is available in the private repository but is not available in the public repository. Readers who would like to have access to the source code may contact the author or directly contact QuviQ.com and Erlang Solutions for their permissions.

ATM simulator This example contains 5 actor classes. It simulates a bank ATM system consisting 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 5.1, cited from [Cesarini, 2011], gives the Finite State Machine that models the behaviour of the front-end of the ATM.

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

Figure 5.2 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 other 4 actors are supervisors for other components. The example is shipped with QuickCheck properties that checks whether events generated by users are correctly handled.

5.1.1.2 Examples from the Akka Documentation

The Akka Documentation [Typesafe Inc. (b), 2012] contains some examples that demonstrate actor programming and supervision in Akka. The author has ported the following examples to check that applications built using TAkka behave similarly to Akka applications.

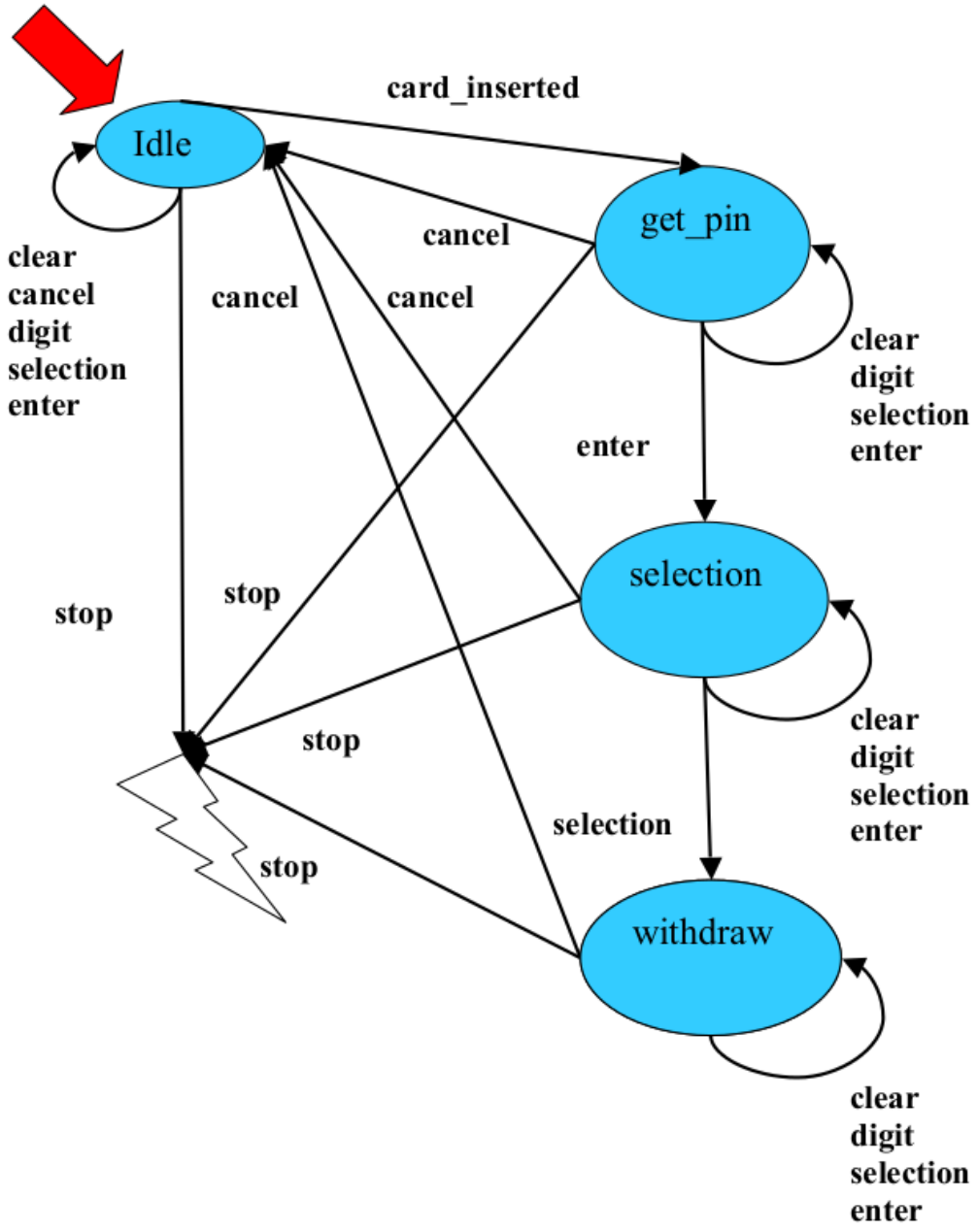


Figure 5.1: Example: ATM – [Arts et al., 2006]

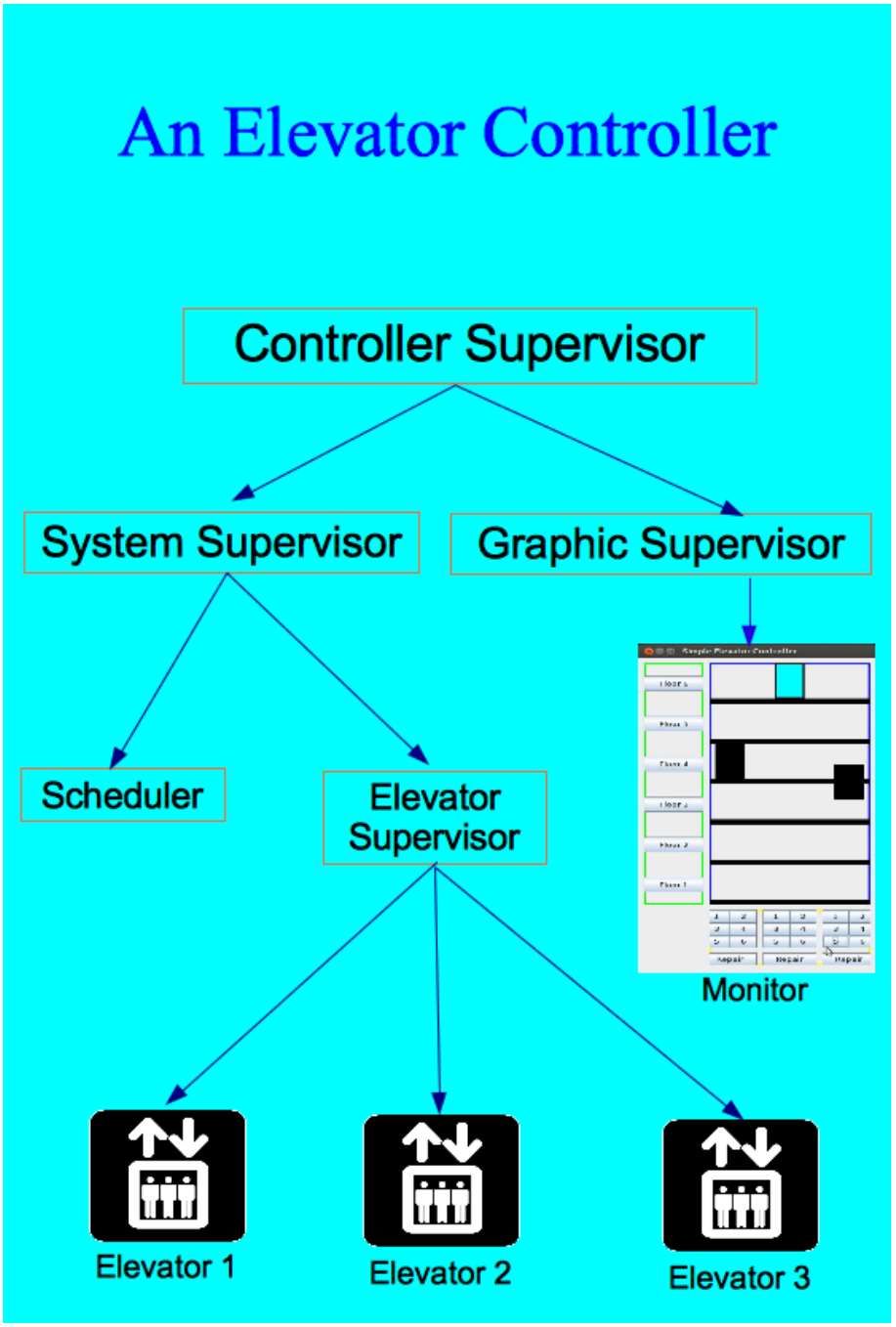


Figure 5.2: Example: Elevator Controller

Ping Pong This example contains 2 actor classes: `Ping` and `Pong`. The two actors send messages to each other 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 2 actor classes. It models the dining philosophers problem [Wikipedia, 2013a] using Finite State Machines (FSMs). The Dining Philosophers problem is one of the classical problems that exhibit synchronization issues in concurrent programming. In the ported version five philosophers sit around a table with five chopsticks interleaved. Each philosopher alternately thinks and eats. Before starting eating, a philosopher needs to hold chopsticks on both sides. At any time, a chopstick can only be held by one philosopher. A philosopher puts down both chopsticks when he finishes eating and thinks for a random period.

Distributed Calculator This example contains 4 actor classes. It demonstrates distributed computation and dynamic behaviour update on the receive function of an actor. The TAkka version of this example is used as a case study in Section 3.12.

Fault Tolerance This example contains 5 actor classes. It models 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.

5.1.1.3 Examples from Other Open Source Projects

The QuviQ examples and the Akka documentation examples are demonstration examples for training purposes. This thesis further ports the following examples from open source projects to enlarge the scope of the test.

Barber Shop This application has 6 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 5 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 4 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 4 actor classes. Gatling [Excilys Group, 2012] is a stress testing tool for web applications. It uses actors and synchronous I/O methods to improve its efficiency. The application is shipped with a tool that reports test results in graphical charts.

Play Core The core library of the Play framework only has 1 actor class. 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 at TypeSafe Inc. and in the Play community. Therefore, this project only ports its core library which is also updated less frequently. Because the original Akka Play is an active project on GitHub, a separate repository is forked for the TAkka version [HE, 2014b] so that updating non-core components is easier.

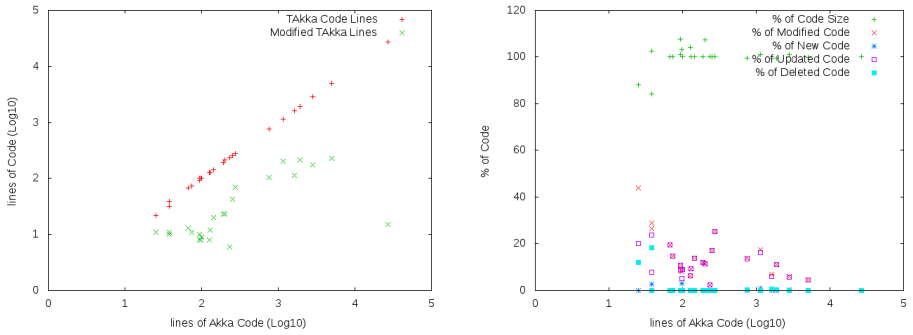
5.1.2 Evaluation Results

5.1.2.1 Code Size

This section investigates whether the type discipline enforced by TAkka restricts the expressibility of Akka. Table 5.1 lists the examples used for expressiveness checks. Medium-seized examples are selected from QuviQ [Arts et al., 2006] and open source Akka projects to ensure that the main requirements for actor programming are not unintentionally neglected. Examples from Quviq are re-implemented using both Akka and TAkka. Examples from Akka projects are re-implemented using TAkka. Following standard practice Fleming and Wallace [1986] and Patterson and Hennessy [2013], the author

Source	Example	Akka Code Lines	TAkka Code Lines	Added Lines	Updated Lines	Deleted Lines
Small Examples	String Processor	25	22	0	5	3
	Supervised Calculator	38	32	1	3	7
	Behaviour Upgrade	38	39	1	9	0
	Fib	74	74	0	11	0
	NQueens	235	235	0	6	0
BenchErl Examples	bang	93	94	0	8	0
	big	93	100	0	10	0
	ehb	201	216	0	23	0
	genstress	129	129	0	12	0
	mbrot	125	130	0	8	0
	parallel	101	101	0	9	0
	ran	98	101	3	5	0
	serialmsg	146	146	0	20	0
QuviQ [Arts et al., 2006]	ATM simulator	1148	1160	12	187	0
	Elevator Controller	2850	2878	7	165	0
Akka Documentation [Typesafe Inc. (b), 2012]	Ping Pong	67	67	0	13	0
	Dining Philosophers	189	189	0	23	0
	Distributed Calculator	250	250	0	43	0
	Fault Tolerance	274	274	0	69	0
Other Open Source Akka Applications	Barber Shop [Zachrison, 2012]	754	751	0	101	3
	EnMAS [Doyle and Allen, 2012]	1916	1909	0	213	7
	Socko Web Server [Imtarnasan and Bolton, 2012]	5024	5017	0	227	0
	Gatling [Excilys Group, 2012]	1635	1623	0	99	12
	Play Core [Typesafe Inc. (c), 2013]	27095	27095	0	15	0
geometric mean		299.82	300.89	-	-	-

Table 5.1: Expressiveness Evaluation



(a) Code Size: Absolute Lines (b) Code Size: Relative Lines

Figure 5.3: Code Size Evaluation

assesses 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 8.5% 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, through many examples neglected the need of handling undefined messages. On average, the total program size of Akka and TAkka applications are almost the same. In most examples, especially in library implementations and some small examples, there is no need to add a new types as an actor is usually designed to handle messages of the same type. In real applications, such as the ATM and the Elevator example, the number of new types added is still relatively small. Figure 5.3 reports the same result in a Scatter chart.

5.1.2.2 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 those examples always pass instances of `HttpRequestEvent` to the `unapply` method and send the returned values to an actor that accepts mes-

sages of `HttpRequestEvent` type. Fortunately, the design flaw is exposed when upgrading the Socko implementation using TAkka.

5.2 Throughput

The Play example [Typesafe Inc. (c), 2013] and the Socko example [Imtarnasan and Bolton, 2012] used in Section 5.1 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 handled requests per unit of 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 Socko.

For a valid HTTP request sent to path `/json`, e.g. the one given in Figure 5.4a, 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 exchange between applications [JSON ORG, 2013]. Figure 5.4b gives an example of an expected HTTP response. The body of the example response, line 7, is the expected JSON message.

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

To avoid pitfalls mentioned in [Amazon.com, Inc., 2012], a FreeBench [HE, 2013] tool was designed and implemented for benchmarking throughputs of HTTP servers. One feature of the FreeBench tool is that it can benchmark web servers deployed at multiple addresses. In the JSON serialization benchmark, the maximum throughput achieved when using the Elastic Load Balancing (ELB) service [Amazon.com, Inc., 2013b] did not obviously increase when more servers were added. In contrast, when all deployed EC2 servers were benchmarked, the total throughput increased slightly. One possible explanation for the above observation is that the benchmark is bounded by the throughput of

```
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 HTTP Request

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 28
Server: Example
Date: Wed, 17 Apr 2013 12:00:00 GMT

{"message":"Hello, World!"}
```

(b) An Example HTTP Response

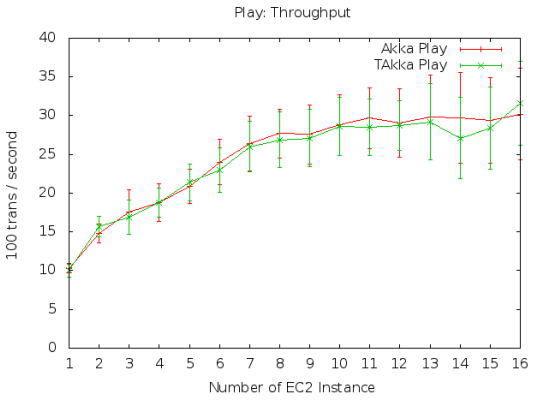
Figure 5.4: Example: JSON serialization Benchmark

ELB, which runs on a micro EC2 instance. 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 a certain number of times. The benchmark results of all tests are sent to a data store which reports a customised statistical summary.

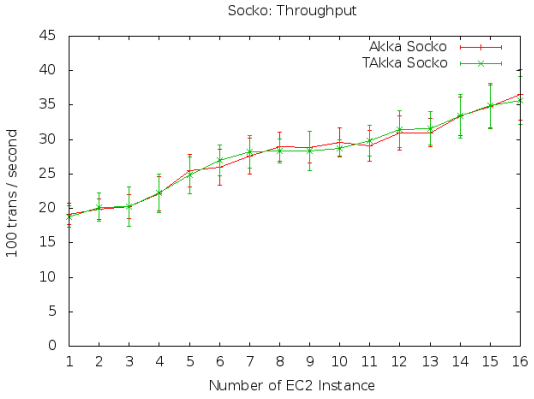
The parameters set in this example were the number of EC2 instances used. For each of the four types of server, the example was tested with up to 16 EC2 instances. For each number of EC2 instances, 10 rounds of benchmarking were executed. In each round, 20 sub-benchmarks were carried out in parallel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

Figure 5.5 summarises the results of the JSON serialization benchmark. It shows the mean and the standard deviation of the throughput in each test. The result shows that web servers built using an Akka-based library and a TAkka-based library have similar throughput. The micro example used in this test does not show a good throughput scalability in both Akka and TAkka versions. It would be more interesting if we can benchmark on a real Akka

web service whose throughput is linear to the number of available servers.



(a)



(b)

Figure 5.5: Throughput Benchmarks

5.3 Efficiency and Scalability

The TAkka library is built on top of Akka so that code for shared features can be re-used. The main sources of overheads in the TAkka implementation are:

i) the cost of adding an additional operational layer on top of Akka code,

ii) the cost of constructing type descriptors,

iii) the cost of transmitting type descriptors in distributed settings, and

iv) the cost of dynamic type checking when registering new typed names.

The upper bounds of costs i) and ii) were assessed by a micro benchmark which assessed the time for 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 5.6, the TAkka implementation runs roughly half as fast as the Akka implementation.

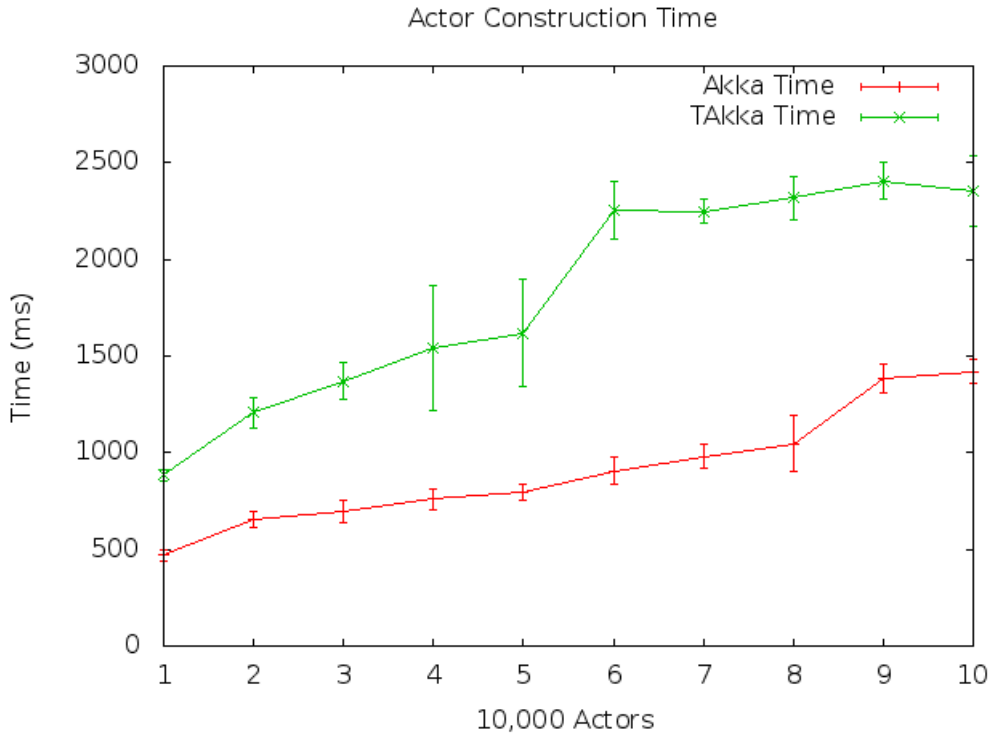


Figure 5.6: Cost of Actor Construction

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 of the latter cost depends on the cost of computations spent on application logic.

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

5.3.1 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. This thesis uses BenchErl examples, which do not involve 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 assembles them and reports the overall elapsed time for the benchmark.

BenchErl examples have similar structure to the MapReduce model [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 in computational resources rather than specific parallelization and scheduling strategies of each example.

5.3.2 Benchmark Examples

5.3.2.1 Ported BenchErl Examples

The following BenchErl examples were ported for comparing the efficiency and scalability of applications 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 send the master process a fixed number of dummy messages. The master process initializes 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 counts 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 bounded by throughput of the master node.

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 with 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 completes 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. Similarly to the `bang` example, the benchmark is likely to be bounded by throughput of the master node.

ehb This benchmark re-implements the hackbench example [Zhang, 2008] originally used for stress testing 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 waits for an acknowledge message. Each sender repeats the process a number of times. When a sender has received all expected replies, it reports to the child actor that it has completed its task. When all senders in the group have completed their tasks, the child process sends a complete message to the master process. The benchmark completes when all child processes have finished 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. When the number of available nodes to share the task of child processes is increased, 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. There are two versions in Erlang, one using the OTP *gen_server* behaviour, the other implementing a simple server-client structure manually. This benchmark ports the version not 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 bounded by throughput of the master node.

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 share roughly the same number of points. The benchmark completes when all child processes have finished 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 creates. 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 sending results to the master 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 logarithmically 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 a 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 the message. The benchmark completes when the master process receives messages from all receivers.

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

5.3.2.2 BenchErl Examples that are Not Ported

The following BenchErl examples are not ported for reasons given in respective 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 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 Similarly to the *ets_test* example, this benchmark also tests 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 either because it uses the ETS table.

timer_wheel Similarly to the *big* example, this benchmark spawns a number of child processes that exchange ping and pong messages. Differently to the *big* example, processes in this example can be configured to await reply messages only for a specified timeout. In cases where no timeout is set, or it is set to a short period, this example is the same as the *big* example. If a timeout is set to a long period, the runtime of this example is bounded by the timeout. For the above reason, this example is not ported.

5.3.3 Benchmark Methodology

5.3.3.1 Testing Environment

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

5.3.3.2 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 know how the required run-time of a BenchErl example changes when more machines are employed. Ideally, for each comparison on the efficiency of an Akka application and its equivalent TAkka application, the only variable should be the number of employed nodes. Nevertheless, the BenchErl examples listed in Section 5.3.2.1 have more parameters to be configured. Experiments for our main interests can be carried out with any parameters, however; for consideration of our secondary interest, parameters for each example were selected according to the following three criteria:

First, except for the RAN example, the runtime of each experiment was constrained to be 40 seconds. The decision was made so that the time to measure each example was acceptable. As will be explained in the next section, each example was tested for a total of 90 rounds of experiments. Another reason for this decision was that the experiment should be able to complete in a reasonably short time when running on a single machine. During the experiment, the author observed some configurations such that an example had a bad performance when run on a single machine but sped up by a factor bigger than the total number of available nodes when run with more nodes. These experiments give interesting results that proved the importance of distributed programming; however, it is desired that the number of nodes be the only independent variable throughout all experiments. Therefore, the benchmarks preferred configurations that neglected the impact of other factors such as garbage collection.

Second, the benchmarks preferred configurations that had more workload for each child process. In a number of tests to determine parameters, the author observed employing more machines only had runtime benefit for those BenchErl examples whose runtime is bounded by the computational tasks rather than the throughput of the only master process.

Third, the benchmarks preferred configurations that had more child processes but did not violate the above two principles. The benchmark was 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, it is started with a small number of child processes. If the child process could have a higher workload by setting other parameters, other parameters were changed until the configuration violated the first criterion. If the number of child processes and the number of available nodes were the only two parameters, or the workload of each child process did not change significantly with other possible configurations, the number of child processes was increased gently until the example took a long time to be completed on a single machine.

Based on the results of trial experiments, the parameters used in each example were as follows:

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

5.3.3.3 Measurement Methodology

After determining the benchmark parameters for each example, the runtime of each program was measured as follows. First, each benchmark contains nine tests that use different numbers of Beowulf nodes. The number of nodes used in the benchmarks were 1, 4, 8, 12, 16, 20 , 24, 28, and 32. Similarly to the Benchmark Harness process in [Blackburn et al., 2006], test results was recorded after a number of dry-runs to warm-up the runtime environment. After the warm-up period, the test was run 10 times. The run time was recorded for later analysis. Following guidance given by Fleming and Wallace [1986] and Patterson and Hennessy [2013], the efficiency of each example using a specific number of nodes is reported by giving the mean and standard deviation of the 10 runs. The speed-up of a benchmark example using n nodes is measured as the proportion of the mean time with one node and the mean time with n nodes. After each test, the runtime environment was cleaned up before changing the number of nodes or switching to another benchmark example.

5.3.4 Evaluation Results

The records of the BenchErl benchmarks are summarised in Figure 5.7. The efficiency results include both the mean runtime and the standard deviation. The scalability results are computed based on the mean runtime. The author observes the following though benchmarking:

Observation 1 In all examples, TAkka and Akka implementations had almost identical run-times and hence have similar scalability. In Figure 5.7, the runtime of Akka benchmarks and TAkka benchmarks often overlay each other. For benchmarks that do not overlay, the difference is less than 10% on average. The scalability of Akka applications and the scalability of their TAkka equivalents appear slightly different because their differences are amplified by their runtime differences 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. These examples do not scale when the number of nodes are greater than a certain number. Examples of this observation are the `Bang` ex-

ample, the `Big` example, and the `Ran` 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, and the `SerialMsg` example.

5.3.5 Additional Benchmarks

5.3.5.1 Fibonacci Numbers

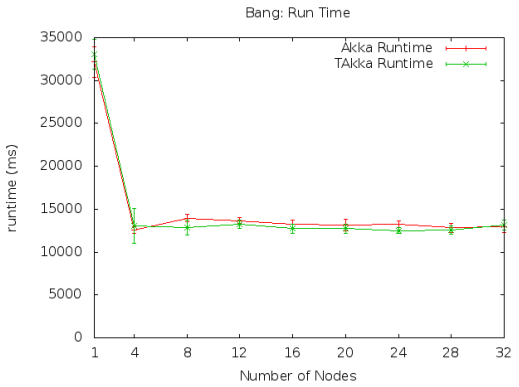
Results given in the last section show that the scalability of BenchErl examples varies. Because BenchErl examples have similar structure and those examples are run on the same environment, the difference in their scalability may lie in differences in their computational tasks. It is expected that the required runtime of a BenchErl example would depend on the time needed for completing the computation task and the time for assembling results. Because the master process is the only one 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 spent on completing parallelized computational tasks and that spent on assembling results, a similar benchmark example was added, where each child process computes the *same* Fibonacci number *sequentially* using the following equation.

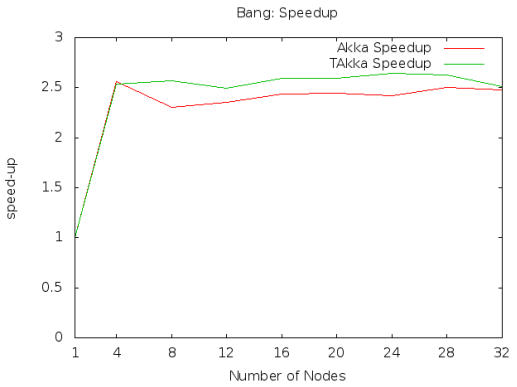
$$f(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2), & \text{if } n >= 2 \end{cases} \tag{5.1}$$

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

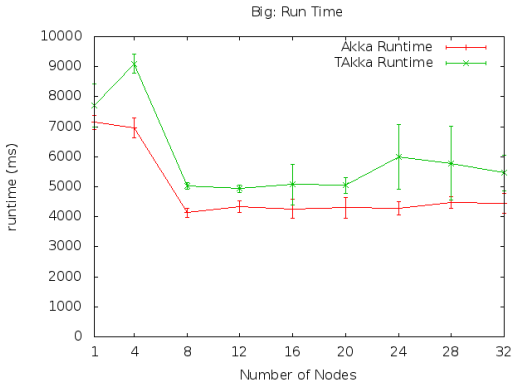
The parameters set in this example are the number of available nodes, the number of child processes, and the value of n in $f(n)$. The author expected that, when setting the number of child processes to a number higher than the number of available nodes, a benchmark with a higher n would give better scalability than those with a lower n. The above expectation is confirmed by the benchmark result reported in Figure 5.8.



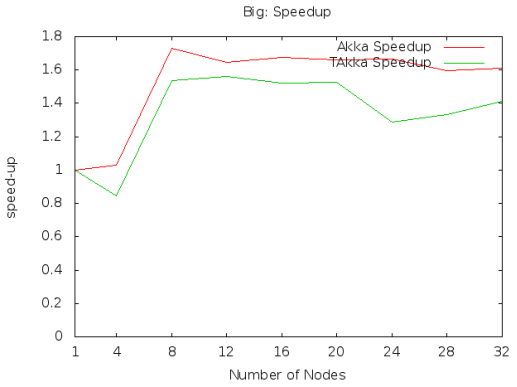
(a) Bang Time



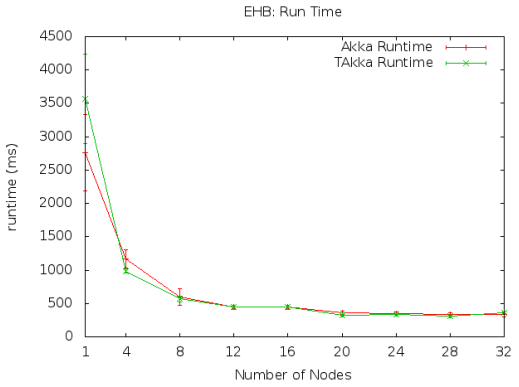
(b) Bang Scalability



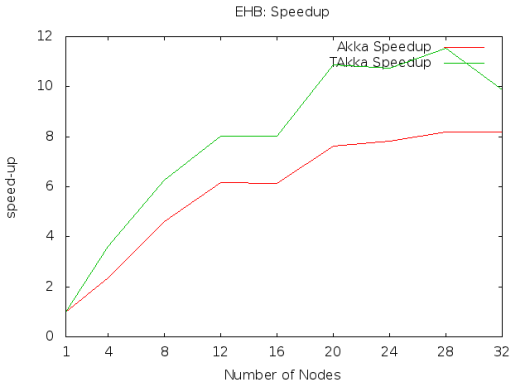
(c) Big Time



(d) Big Scalability



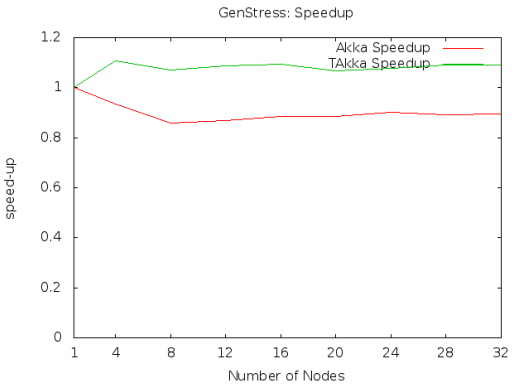
(e) EHB Time



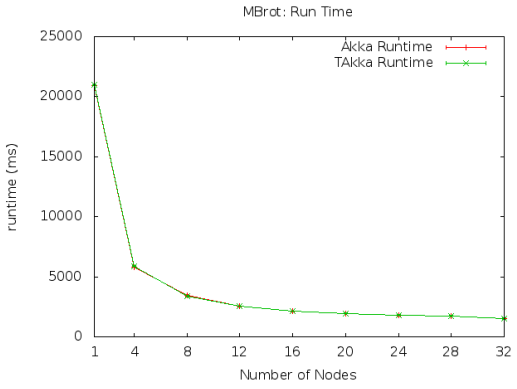
(f) EHB Scalability

Figure 5.7: Runtime and Efficiency Benchmarks

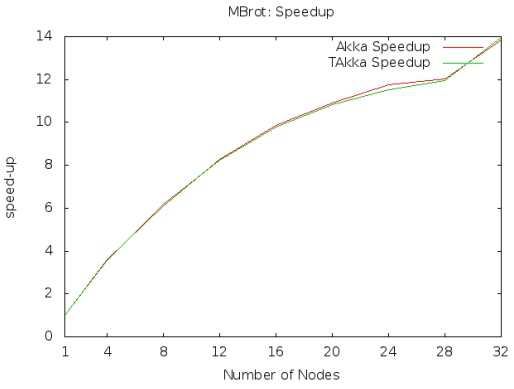
(g) GenStress Time



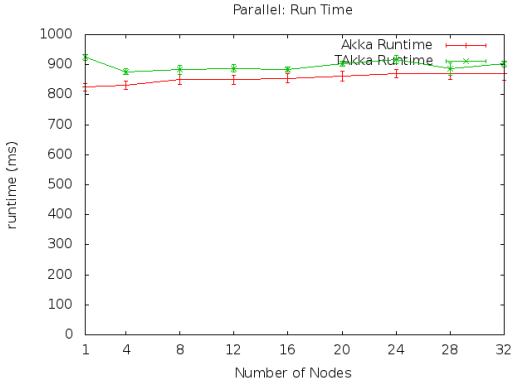
(h) GenStress Scalability



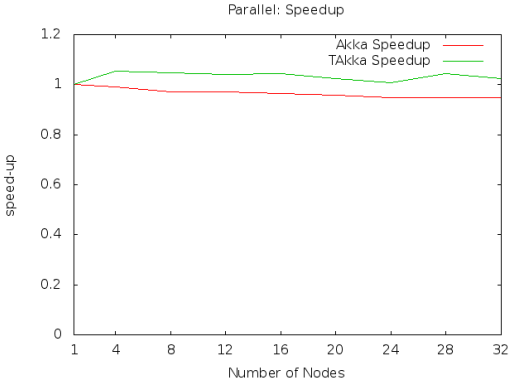
(i) MBrot Time



(j) MBrot Scalability

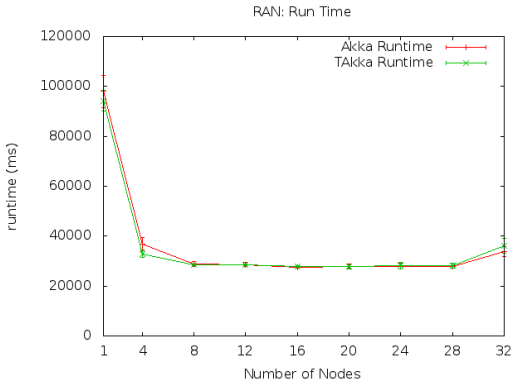


(k) Parallel Time

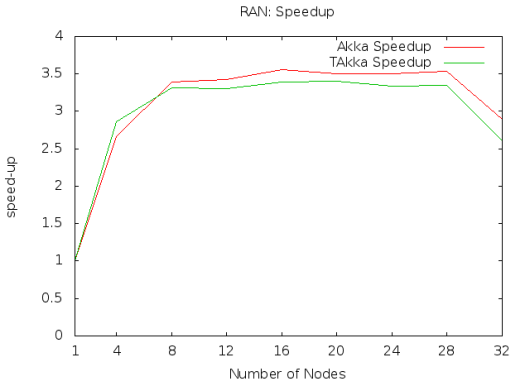


(l) Parallel Scalability

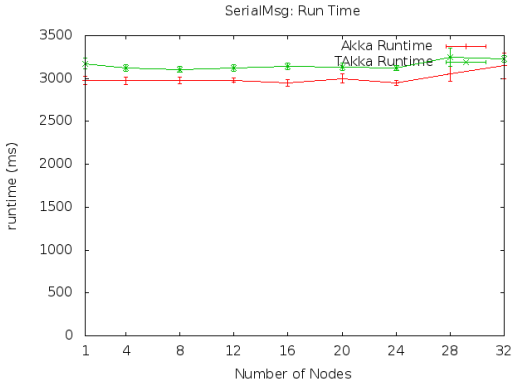
Figure 5.7: Runtime and Efficiency Benchmarks



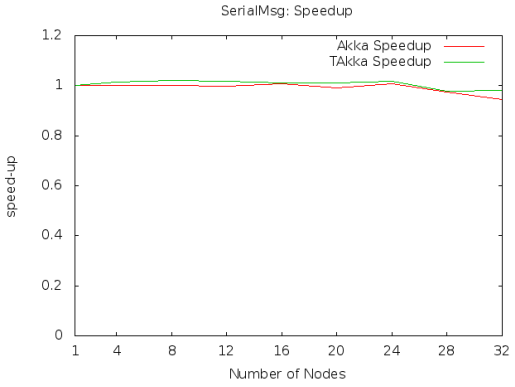
(m) RAN Time



(n) RAN Scalability

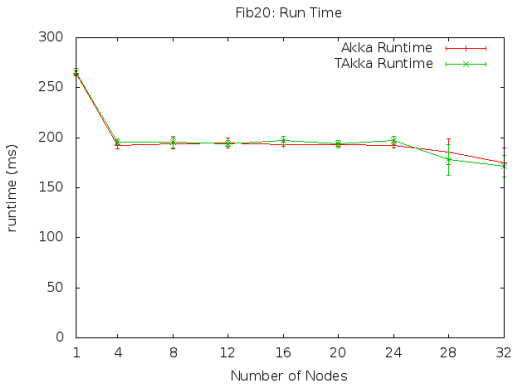


(o) SerialMsg Time

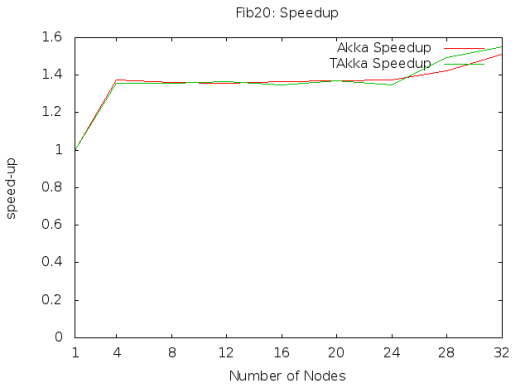


(p) Serial Scalability

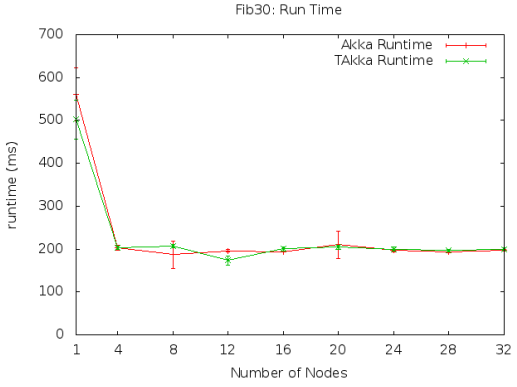
Figure 5.7: Runtime and Efficiency Benchmarks



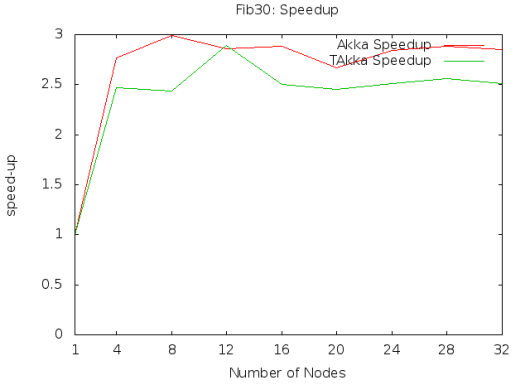
(a) Fib20 Time



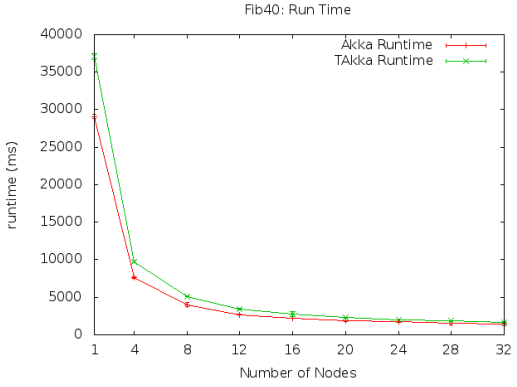
(b) Fib20 Scalability



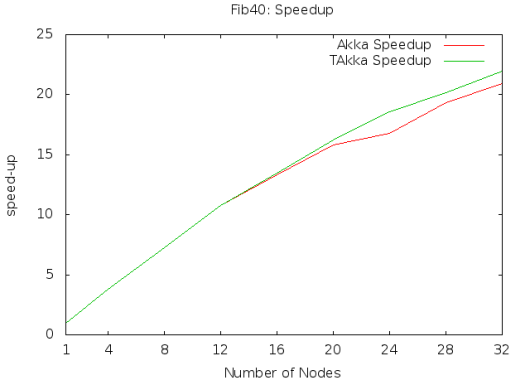
(c) Fib30 Time



(d) Fib30 Scalability



(e) Fib40 Time



(f) Fib40 Scalability

Figure 5.8: Benchmark:Parallel Fibonacci Numbers

5.3.5.2 MBrot with different image sizes

The result of the Fibonacci benchmark provides evidence that the scalability of a distributed application can be bounded by the throughput of the master node. The same conclusion is obtained when the image size of the MBrot example is changed. The MBrot example has a quadratic complexity to the image size. In the BenchErl example, the image size is set to 6000 pixels × 6000 pixels so that more time is spent on computation than message sending. Figure 5.9 reports the results when the image size is set to 10 pixels × 10 pixels, 1000 pixels × 1000 pixels, and 6000 pixels × 6000 pixels. As expected, the result is similar to the result of the Fibonacci benchmark.

5.3.5.3 The N-Queens Problem

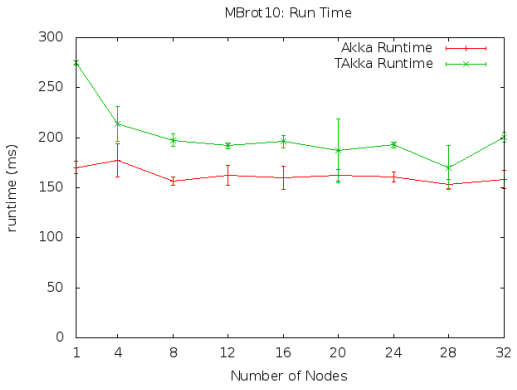
In the BenchErl examples and the Fibonacci example, child processes are asked to execute the same computation a number of times (Section 5.3.1). In contrast, distributed and cluster computing techniques are often used to solve a computationally expensive task by distributing sub-tasks to independent nodes. To simulate such a scenario, another benchmark, N-Queens Puzzle, is added.

The N-Queen Puzzle [Wikipedia, 2014] looks for *all* solutions of placing n queens on an $n \times n$ chessboard such that no two queens share the same row, column, or diagonal. In the benchmark, a master node first uses a width-first backtracking algorithm to expand the search space. If the number of candidate partial solutions is greater than twice the available nodes, the master node sends partial solutions to those nodes which use a depth-first backtracking algorithm to find all possible solutions of each candidate partial solution. In this example, two solutions are considered distinct if they differ only in symmetry operations.

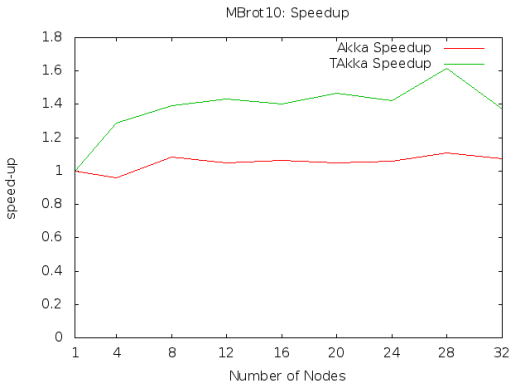
Finding all solutions of an N-Queen Puzzle is an NP-hard problem. Therefore, a suitable n makes the problem a good benchmark to demonstrate the advantage of cluster and distributed programming. Figure 5.10 reports the result when n is set to 14. The value of n is chosen according to the same criteria for BenchErl benchmarks as stated in Section 5.3.3.2. The result shows that both the Akka and TAkka implementation have good scalability and similar efficiency.

5.4 Assessing System Reliability and Availability

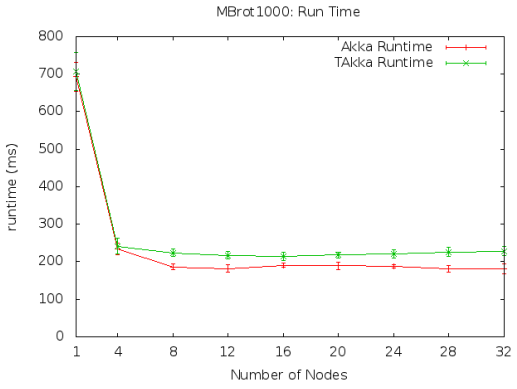
The supervision tree principle is adopted by Erlang and Akka users in the hope of improving the reliability of software applications. Apart from the



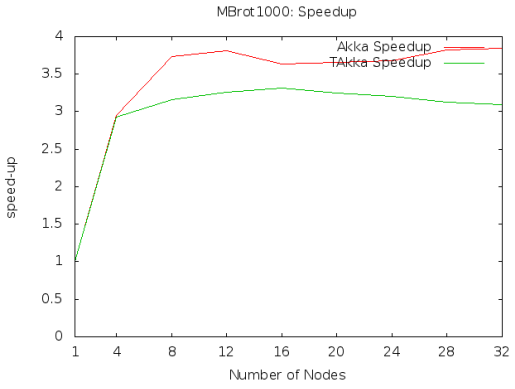
(a) MBrot10 Time



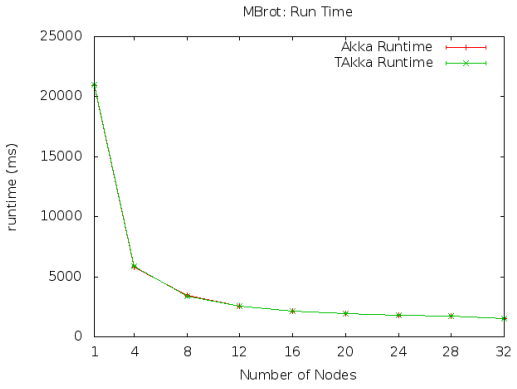
(b) MBrot10 Scalability



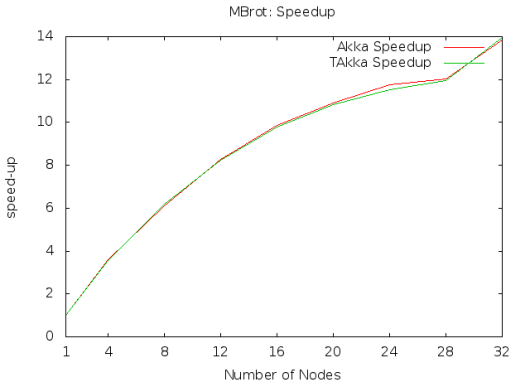
(c) MBrot1000 Time



(d) MBrot1000 Scalability

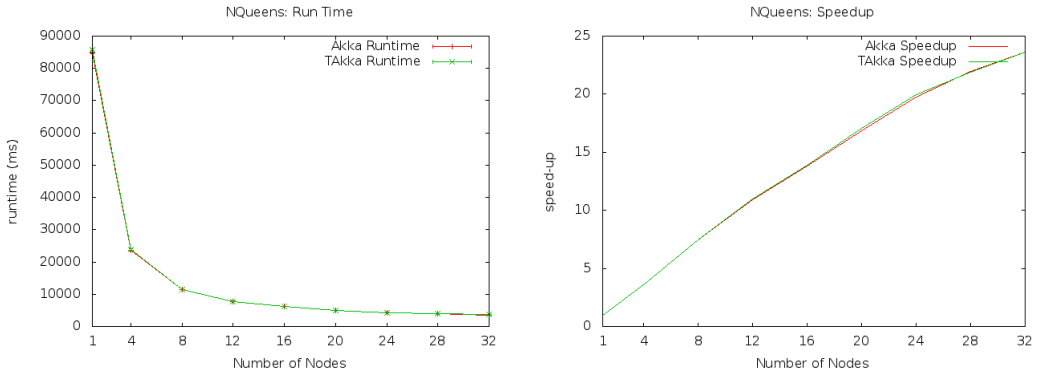


(e) MBrot6000 Time



(f) MBrot6000 Scalability

Figure 5.9: Benchmark: MBrot



(a) N-Queens Time (b) N-Queens Scalability

Figure 5.10: Benchmark: N-Queens Puzzle

reported nine "9"s reliability of Ericsson AXD 301 switches [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 5.4.3) show that failures occur during the test. On the other hand, any failed actors are restored, and hence appropriately supervised applications (Section 5.4.4) pass Chaos Monkey tests.

5.4.1 Chaos Monkey and Supervision View

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

Random This is the default mode of Chaos Monkey. It randomly choose one of the other modes in each run.

Exception This mode simulates the case that an exception is raised from an actor. The randomly picked victim actor raises an exception from a user-defined set of exceptions.

Kill This mode simulates the case that a recoverable failure is occurred inside an actor. A `Kill` message is sent to a randomly picked victim actor to see if it can be restarted by its supervisor.

PoisonKill This mode simulates that case that an unidentifiable failure is occurred inside an actor. A `PoisonKill` message is sent to a randomly picked victim actor. The actor is terminated permanently and cannot be restarted by its supervisor. It tests whether the application has other failure recovery mechanism for that actor. Being supervised is sufficient for most actors. However, for some critical actors, having additional assurance might be required in practice.

NonTerminate This mode simulates network congestion or a design flow of actor implementation. A randomly picked actor runs into an infinite loop and consumes system resources but cannot process any messages. A robust system should be able to detect such flow or network congestion, redirect further messages to a new actor, and try to kill the problematic actor.

Figure 5.11 gives the API and the core implementation of TAkka Chaos Monkey. A user sets up a Chaos Monkey test by initializing a `ChaosMonkey` instance, defining the test mode, and scheduling the interval between each run. In each run, the `ChaosMonkey` instance sends a randomly picked actor a special message. Upon receiving a Chaos Monkey message, a TAkka actor executes a piece of problematic code as described above. `PoisonPill` and `Kill` are handled by `systemMessageHandler` and can be overridden (Figure 3.2). `ChaosException` and `ChaosNonTerminate`, on the other hand, are handled by the TAkka library and cannot be overridden.

5.4.2 Supervision View

To dynamically monitor changes in supervision trees, the author designed and implemented a Supervision View library. In a supervision view test, an instance of `ViewMaster` periodically sends request messages to interested actors. When the request message is received, an active TAkka actor replies its status to the `ViewMaster` instance and passes the request message to its children. The status message includes its actor path, paths of its children, and the time when the

```
package takka.chaos
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 5.11: TAkka Chaos Monkey

reply is sent. The `ViewMaster` instance records status messages and passes them to a visualizer, which will analyze and interpret changes in the tree structure during the testing period.

A view master is initialized by calling one of the apply methods of the `ViewMaster` object as given in Figure 5.12. 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 `Actor[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 passes the `SupervisionViewRequest` message to its children. The `date` value in a `SupervisionViewResponse` message is the same as the `date` value in the corresponding `SupervisionViewRequest` message. Finally, the master actor of the view master records all replies in a hash map from `Date` to `TreeSet[NodeRecord]`, and sends the record to an appropriate drawer on request.

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

Figure 5.12: Supervision View

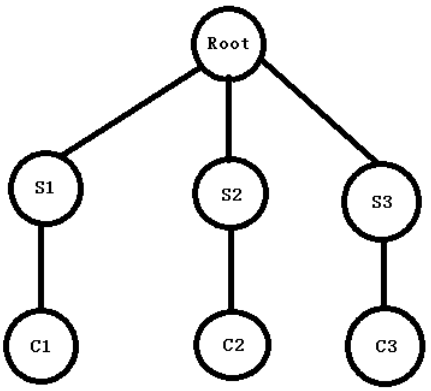
5.4.3 A Partly Failed Safe Calculator

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

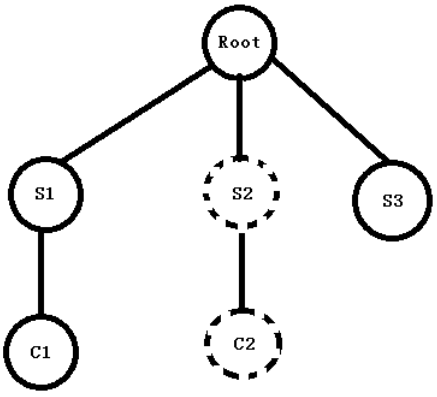
Figure 5.13 gives a visual interpretation of the textual test result. The 3 graphs show the tree structures at the beginning, 15 seconds and 30 seconds of the test. Figure 5.13a shows that the application initialized three safe calculators as described. In Figure 5.13b, S2 and its child are marked as dashed circles because it takes the view master more than 5 seconds to receive their responses. From the test result itself, a user cannot tell whether the delay is due to a blocked calculation or network congestion. Comparing against Figure 5.13a, the child of S3 is not shown in Figure 5.13b and Figure 5.13c 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 5.13c. S1 and its child appear in all three Figures because either they never fail during the test or they are recovered from failures within a short time.

5.4.4 BenchErl Examples with Different Supervisor Strategies

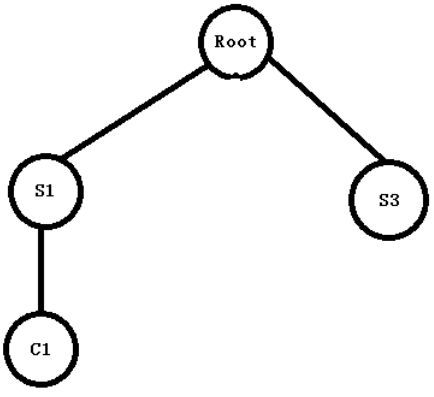
To test the behaviour of applications with internal states under different supervisor strategies, the author applied the `OneForOne` supervisor strategy with different `Directives` (Figure 3.14) to the 8 BenchErl examples and tested them using Chaos Monkey and Supervision View. The master node of each BenchErl test was initialized with an internal counter. The internal counter decreased when the master node received finishing messages from its children. The test application stopped when the internal counter of the master node reached 0.



(a)



(b)



(c)

Figure 5.13: Supervision View Example

The Chaos Monkey test is set with the `Kill` mode and randomly killed a victim actor every second. When the `Escalate` directive is applied to the master node, the test stops as soon as the first `Kill` message is sent from the Chaos Monkey test. When the `Stop` directive is applied, the application does not stop and, eventually, the supervision view test only receives messages from the master node. When the `Restart` directive is applied, the application does not stop but the Supervision View test receives messages from the master node and its children. When the `Resume` directive is applied, all tests stop eventually with a longer run-time compared to tests without Chaos Monkey and Supervision View.

5.5 Summing Up

This chapter confirms that TAkka can detect type errors without bringing in obvious overheads. Firstly, 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 was found in the Socko framework. The case study in Section 5 shows that TAkka has the advantage of solving the type pollution problem. Secondly, web servers built using Akka and TAkka reach similar throughput when the same number of EC2 instances are used. Thirdly, BenchErl benchmark examples written in Akka and TAkka have similar efficiency and scalability when running on a 32 node Beowulf cluster. The additional benchmark examples in Section 5.3.5 provide evidences that the scalability of an application depends on the ratio of the cost of parallelized computational tasks and the cost of throughput bounded communications. Lastly, TAkka provides a ChaosMonkey library and a Supervision View library for assessing the correctness of applications built using TAkka.

Chapter 6

Summary and Future Work

The main goal of this thesis is the development of a library that combines the advantages of type checking and the supervision principle. The aim is to contribute to the construction of reliable distributed applications using type-parameterized actors and supervision trees. Aside from the TAkka library itself, this thesis has 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. This chapter reviews the research results presented in the thesis, suggests future research topics, and concludes.

6.1 Overview of Contributions

6.1.1 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 look 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 equiv-

alents (evolution) rather than require providing type parameters everywhere (revolution). The above property is analogous to adding generics to Java programs.

Compared with Akka, the TAkka library avoids the type pollution problem straightforwardly. The type pollution problem, discussed in Section 3.10, refers to the situation where a user can send a service message not expected from him or her 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. A demonstration example shows that avoiding the type pollution problem in TAkka is as simple as publishing a service as having different types when it is used by different parties.

6.1.2 A Library Evaluation Framework

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

By porting existing small and medium sized Erlang and Akka applications, results in Section 5.1 and 5.3 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. The TAkka library is expected to have 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.

6.2 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 primary evaluations show that the TAkka library can prevent some errors without bringing obvious overheads compared with equivalent Akka applications. As actors and supervision trees are widely used in the development of distributed applications nowadays, the author believes that there is great potential for the TAkka library. Much more can be done to make TAkka more usable, as well as to further the goal of making the building of reliable distributed applications easier.

6.2.1 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 resulting 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 the 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 allows the author 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, the author believes 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]. The author believes that the techniques used in this thesis can help the design of the future versions of Cloud Haskell.

6.2.2 Benchmark Results from Large Real Applications

This thesis compares TAkka with Akka with regards to several dimensions by porting small and medium sized applications. Most of selected examples are

from open source projects. The author gratefully acknowledges the RELEASE team for giving us access to the source code of the BenchErl benchmark examplesl; Thomas Arts from QuviQ.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 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.

6.2.3 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?

6.2.4 Measuring and Predicting System Reliability

Due to the nature of software development, the library itself cannot guarantee the reliability of applications built using it; nor can the achieved high reliability of 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 following.

The first approach is measuring the target system as a black-box. Unfortunately, Littlewood and Strigini [1993] show that even long term failure-free observation itself does not mean that the software system will achieve 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 nodes and simple supervisor nodes. 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 it not be?
- 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?

6.3 Conclusion

The author believes that the demand 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. The author is 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. The author expects similar results can be obtained in other actor libraries.

Appendix A

Akka and TAkka API

```
package akka.actor
abstract class ActorRef
  def !(message: Any):Unit
trait  Actor
  def receive:PartialFunction[Any, Unit]
  val self:  ActorRef
  private val context:  ActorContext
  var supervisorStrategy: SupervisorStrategy
trait ActorContext
  def actorOf(props: Props): ActorRef
  def actorOf(props: Props, name: String): ActorRef
  def actorFor(path: String): ActorRef
  def setReceiveTimeout(timeout: Duration): Unit
  def become(behavior: PartialFunction[Any, Unit],
             discardOld:  Boolean = true):  Unit
def unbecome():  Unit
final case class Props(deploy:  Deploy, clazz:  Class[_],
                       args:  immutable.Seq[Any])
object Props extends Serializable
  def apply(creator: =>Actor): Props
  def apply(actorClass: Class[_ <: Actor]): Props
  def apply[T <: Actor]() (implicit arg0: Manifest[T]): Props
abstract class SupervisorStrategy
case class OneForOneStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
case class OneForAllStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
```

Figure A.1: Akka API

```
package takka.actor

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

abstract class Actor[M:Manifest] extends akka.actor.Actor
  def typedReceive:M=>Unit
  val typedSelf:ActorRef[M]
  private val typedContext:ActorContext[M]
  var supervisorStrategy: SupervisorStrategy

abstract class ActorContext[M:Manifest]
  def actorOf [Msg] (props: Props[Msg])
            (implicit mt:  Manifest[Msg]): ActorRef[Msg]
  def actorOf [Msg] (props: Props[Msg], name: String)
            (implicit mt:Manifest[Msg]): ActorRef[Msg]
  def actorFor [Msg] (path: String)
            (implicit mt:Manifest[Msg]):  ActorRef[Msg]
  def setReceiveTimeout(timeout: Duration): Unit
  def become[SupM >:  M](behavior: SupM=>Unit)
          (implicit smt:Manifest[SupM]):ActorRef[SupM]

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

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

object Props extends Serializable
  def apply[T](creator: => Actor[T]): Props[T]
  def apply[T](actorClass: Class[_<: Actor[T]]):Props[T]
  def apply[T, A<:Actor[T]] (implicit arg0: Manifest[A]): Props[T]

abstract class SupervisorStrategy
case class OneForOneStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
case class OneForAllStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
```

Figure A.2: TAkka API

Appendix B

Scala Join (version 0.3) User Manual

This Appendix presents the Scala Join library implemented by the author as an exercise to understand distributed channel-based communication. The Scala Join Library (version 0.3) is implemented from scratch, overcomes some limitations of the scala joins library implemented by Haller and Van Cutsem [2008]. Main advantages of this library are: (i) providing a uniform *and* operator, (ii) supporting pattern matching on messages, (iii) supporting theoretically unlimited numbers of join patterns in a single join definition (iv) using a simpler structure for the **Join** class, and most importantly, (v) supporting communications on distributed join channels.

B.1 Using the Library

B.1.1 Sending messages via channels

An elementary operation in the join calculus is sending a message via a channel. A channel could be either asynchronous or synchronous. At the caller's side, sending a message via an asynchronous channel has no obvious effects in the sense that the program will always proceed. By contrast, when a message is sent via a synchronous channel, the current thread will be suspended until a result is returned.

To send a message m via channel c, users simply apply the message to the channel by calling $c(m)$. For the returned value of a synchronous channel, users may want to assign it to a variable so that it could be used later. For example,

```
1 val v = c(m) // c is a synchronous channel
```

B.1.2 Grouping join patterns

A join definition defines channels and join patterns. Users define a join definition by initializing the **Join** class or its subclass. It is often the case that a join definition should be globally *static*. If this is the case, it is a good programming practice in Scala to declare the join definition as a *singleton object* with the following idiom:

```
object join_definition_name extends Join{
  //channel declarations
  //join patterns declaration
}
```

A channel is a singleton object inside a join definition. It extends either the **AsyName**[**ARG**] class or the **SynName**[**ARG**, **R**] class, where **ARG** and **R** are generic type parameters. Here, **ARG** indicates the type of channel parameter whereas **R** indicates the type of return value of a synchronous channel. The current library only supports unary channels, which only take one parameter. Fortunately, this is sufficient for constructing nullary channels and channels that take more than one parameter. A nullary channel could be encoded as a channel whose argument is always **Unit** or any other constants. For a channel that takes more than one parameter, users could pack all arguments in a tuple.

Once a channel is defined, we can use it to define a join pattern in the following form

```
case <pattern> => <action>
```

The <*pattern*> at the left hand side of ⇒ is a set of channels and their formal arguments, connected by the infix operator *and*. The <*action*> at the right hand side of ⇒ is a sequence of Scala statements. Formal arguments declared in the <*pattern*> must be pairwise distinct and might be used in the <*action*> part. In addition, each join definition accepts one and only one group of join patterns as the argument of its *join* method. Lastly, like most implementations for the join calculus, the library does not permit multiple occurrences of the same channel in a single join pattern. On the other side, using the same channel in an arbitrary number of different patterns is allowed.

We conclude this section by presenting a sample code that defines and uses join patterns.

```
import join._
import scala.concurrent.ops._ // spawn
object join_test extends App{// for scala 2.9.0 or later
 object myFirstJoin extends Join{
   object echo extends AsyName[String]
   object sq extends SynName[Int, Int]
   object put extends AsyName[Int]
   object get extends SynName[Unit, Int]

   join{
     case echo("Hello") => println("Hi")
     case echo(str) => println(str)
     case sq(x) => sq reply x*x
     case put(x) and get(_) => get reply x
  }
 }

 spawn {
   val sq3 = myFirstJoin.sq(3)
   println("square(3) = "+sq3)
 }
 spawn { println("get: "+myFirstJoin.get()) }
 spawn { myFirstJoin.echo("Hello"), myFirstJoin.echo("Hello World") }
 spawn { myFirstJoin.put(8) }
}
```

Listing B.1: Example code for defining join patterns (join_test.scala)

One possible result of running the above code is:

```
>scalac join_test.scala
>scala join_test
square(3) = 9
Hi
Hello World
get: 8
```

B.1.3 Distributed computation

With the distributed join library, it is easy to construct distributed systems on the top of a local system. This section explains additional constructors in the distributed join library by looking into the code of a simple client-server system, which calculates the square of an integer on request. The server side code is given at Listing B.2 and the client side code is given at Listing B.3.

```
import join._
object Server extends App{
  val port = 9000
  object join extends DisJoin(port, 'JoinServer){
    object sq extends SynName[Int, Int]
    join{ case sq(x) => println("x:"+x); sq reply x*x }
    registerChannel("square", sq)
  }
  join.start()
}
```

Listing B.2: Server.scala

```
object Client{
  def main(args: Array[String]) {
    val server = DisJoin.connect("myServer", 9000, 'JoinServer)
    //val c = new DisSynName[Int, String]("square", server)
    //java.lang.Error: Distributed channel initial error:
    //       Channel square does not have type Int => java.lang.String
        ...
    val c = new DisSynName[Int, Int]("square", server)//pass
    val x = args(0).toInt
    val sqr = c(x)
    println("sqr("+x+") = "+sqr)
    exit()
}}
```

Listing B.3: Client.scala

```
> scala ServerTest
Server 'JoinServer Started...'
>scala Client 5
x:5
sqr(5) = 25
>scala Client 7
x:7
sqr(7) = 49
```

In Server.scala, we constructed a distributed join definition by extending class **DisJoin**(**Int**, **Symbol**). The integer is the port where the join definition will listen and the symbol is used to identify the join definition. The way to declare channels and join patterns in **DisJoin** is the same as the way in **Join**. In addition, channels which might be used at remote site are registered with a memorizable string. At last, different from initializing a local join definition, a distributed join definition has to be explicitly started.

In Client.scala, we connect to the server by calling DisJoin.connect. The first and second arguments are the hostname and port number where the remote

join definition is located. The last argument is the name of the distributed join definition. The hostname is a String which is used for the local name server to resolve the IP address of a remote site. The port number and the name of join definition should be exactly the same as the specification of the distributed join definition.

Once the distributed join definition, server, is successfully connected, distributed channels could be initialized as instances of DisAsyName[ARG](channel_name, server) or DisSynName[ARG, R](channel_name, server). Using an unregistered channel name or declaring a distributed channel whose type is inconsistent with its referring local channel will raise a run-time exception during the channel initialization. In later parts of the program, the client is free to use distributed channels to communicate with the remote server. The way to invoke distributed channels and local channels are the same.

B.2 Implementation Details

B.2.1 Case Statement, Extractor Objects and Pattern Matching in Scala

In Scala, a partial function is a function with an additional method: *isDefinedAt*, which will return *true* if the argument is in the domain of this partial function, or *false* otherwise. The easiest way to define a partial function is using the case statement. For example,

```
scala> val myPF : PartialFunction[Int,String] = {
     | case 1 => "myPF apply 1"
     | }
myPF: PartialFunction[Int,String] = <function1>

scala> myPF.isDefinedAt(1)
res1: Boolean = true

scala> myPF.isDefinedAt(2)
res2: Boolean = false

scala> myPF(1)
res3: String = myPF apply 1

scala> myPF(2)
scala.MatchError: 2
  at $anonfun$1.apply(<console>:5)
  ...
```

In addition to basic values and case classes, the value used between *case* and ⇒ could also be an instance of an *extractor object*: object that contains an *unapply* methodEmir et al. [2007]. For example,

```
scala> object Even {
     |   def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else
        None
     | }
defined module Even

scala>  42 match { case Even(n) => Console.println(n) } // prints 21
21

scala>  41 match { case Even(n) => Console.println(n) } // prints 21
scala.MatchError: 41
  ...
```

In the above example, when a value, say x, attempts to match against a pattern, $Even(n)$, the method $Even.unapply(x)$ is invoked. If $Even.unapply(x)$ returns $Some(v)$, then the formal argument n will be assigned with the value v and statements at the right hand side of ⇒ will be executed. By contrast, if $Even.unapply(x)$ returns *None*, then the current case statement is considered not matching the input value, and the pattern examination will move towards the next case statement. If the last case statement still does not match the input value, then the whole partial function is not defined for the input. Applying a value outside the domain of a partial function will rise a *MatchError*.

B.2.2 Implementing local channels

Both asynchronous channel and synchronous channel are subclasses of trait *NameBase*. The reason why we introduced this implementation free trait is that, although using generic types to restrict the type of messages pending on a specific channel is important for type safety, a uniform view for asynchronous and synchronous channels simplifies the implementation at many places. For example, the three methods listed in Listing B.4 are common between those two kinds of channels and are important for the implementation of Join and DisJoin class.

```
trait NameBase{ // Super Class of AsyName and SynName
  def argTypeEqual(t:Any):Boolean
  def pendArg(arg:Any):Unit
  def popArg():Unit
}
```

Listing B.4: The NameBase trait

```
class AsyName[Arg](implicit owner: Join, argT:ClassManifest[Arg])
    extends NameBase{
  var argQ = new Queue[Arg] //queue of arguments pending on this name

  override def pendArg(arg:Any):Unit = {
    argQ += arg.asInstanceOf[Arg]
  }

  def apply(a:Arg) :Unit = synchronized {
    if(argQ.contains(a)){ argQ += a }
    else{
      owner.trymatch(this, a) // see if the new message will trigger
          any pattern
    }
  //other code
}
```

Listing B.5: Code defines local asynchronous channel

Asynchronous channel is implemented as Listing B.5. The implicit argument *owner* is the join definition where the channel is defined. The other implicit argument, **argT**, is the descriptor for the run time type of **Arg**. Although **argT** is a duplicate information for **Arg**, it is important for distributed channels, whose erased type parameter might be declared differently between different sites. We postpone this problem until §B.2.4.

As shown in the above code, an asynchronous channel contains an argument queue whose element must have generic type **Arg**. Sending a message via a channel is achieved by calling its *apply* method, so that $c(m)$ could be written instead of c.apply(m) for short in Scala. Based on the linear assumption that no channel should appear more than once in a join pattern, reduction is possible only when a new message value is pending on a channel. Therefore, if the new message has the same value as another pended message, it should be attached to the end of the message queue; Otherwise, the join definition will be notified to perform a pattern checking and fire a possible pattern, if there is one.

As listing B.6 shows, in addition to firing a pattern or pending a message to the message queue, an invocation on synchronous channel also needs to return a result value to the message sender. Since many senders may be waiting for a return value at the same time, for each reply invocation, the library need to work out which message the result is replied for. To this end, messages with the same value is tagged with different integers. The library uses msgTags to store the message that matches current fireable pattern. When a reply method is called, the channel inserts a integer-message pair and its corresponding reply

value to the result queue and notifies all fetch threads that are waiting for a reply. With the help of the synchronized method, only one thread could attempt to fetch the reply value at a time.

```
class SynName[Arg, R](implicit owner: Join, argT:ClassManifest[Arg],
    resT:ClassManifest[R])
 extends NameBase{
  var argQ = new Queue[(Int,Arg)] // argument queue
  var msgTags = new Stack[(Int,Arg)] // matched messages
  var resultQ = new Queue[((Int,Arg), R)] // results

  private object TagMsg{
    val map = new scala.collection.mutable.HashMap[Arg, Int]
    def newtag(msg:Arg):(Int,Arg) = {
      map.get(msg) match {
        case None =>
          map.update(msg,0)
          (0,msg)
        case Some(t) =>
          map.update(msg, t+1)
          (t+1, msg)
   }}}
  def pushMsgTag(arg:Any) = synchronized {
    msgTags.push(arg.asInstanceOf[(Int,Arg)])
  }
  def popMsgTag:(Int,Arg) = synchronized {
    if(msgTags.isEmpty) { wait(); popMsgTag }
    else{ msgTags.pop }
  }
  def apply(a:Arg) :R = {
    val m = TagMsg.newtag(a)
    argQ.find(msg => msg._2 == m._2) match{
      case None => owner.trymatch(this, m)
      case Some(_) => argQ += m
    }
    fetch(m)
  }
  def reply(r:R):Unit = spawn {synchronized {
    resultQ.enqueue((msgTags.pop, r))
    notifyAll()
  }}
  private def fetch(a:(Int,Arg)):R = synchronized {
    if (resultQ.isEmpty || resultQ.front._1 != a){
      wait(); fetch(a)
    }else{ resultQ.dequeue()._2 }
  }
  // other code
}
```

Listing B.6: Code defines local synchronous channel

B.2.3 Implementing the join pattern using extractor objects

The unapply method for local synchronous channel In this library, join patterns are represented as a partial function. To support join patterns and pattern matching on message values, the library provides the unapply method for local channels. The unapply method for synchronous channel is given in Listing B.8. The unapply method for asynchronous channel is almost the same as the synchronous version, except that it does not need to deal with message tags.

Listing B.7 gives the core of the unapply method of synchronous channel. The five parameters sent to the unapply method are:

(i) *nameset*: channels that could trigger the first fireable pattern.

(ii) *pattern*: the join pattern itself.

(iii) $fixedMsg$: a map from channels to corresponding message values. If the current channel is a key of the map, the unapply method returns its mapped value.

(iv) *dp*: an integer indicates the depth of pattern matching. The *dp* is useful for optimizations and debugging.

(v) *bandedName*: a banded channel name. If the current channel is the same as the *bandedName*, the unapply method returns None.

When a channel is asked to select a message that could trigger a *pattern*, it first check rule (iii) and (v). If neither rule applies, the channel returns the first message that matches the pattern and adds this channel to the *nameset*, if such a message exists. We consider a message of the current channel triggers a join pattern if the join pattern cannot be fired without the presence of messages on the current channel and will be fired when that message is bound to the current channel.

```
case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
          fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase)
             => {
       //other code
       if (this == banedName) {return None}
       if(fixedMsg.contains(this)){
         Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2)
       }else{

         def matched(m:(Int,Arg)):Boolean = {
         //    pattern cannot be fired without the presence of message
             on current channel
         //and pattern can be fired when m is bound to the current
             channel
              (!(pattern.isDefinedAt(nameset, pattern, fixedMsg+((this,
                 m)), dp+1, this))
           && (pattern.isDefinedAt((nameset, pattern, fixedMsg+((this,
              m)), dp+1, banedName))))
         }

       var returnV:Option[Arg] = None

       argQ.span(m => !matched(m)) match {
          case (_, MutableList()) => { // no message pending on this
              channel may trigger the pattern
            returnV = None
          }
          case (ums, ms) => {
            val arg = ms.head // the message could trigger a pattern
            nameset.add(this)
            if(dp == 1) {pushMsgTag(arg)}
            returnV = Some(arg)
          }
       }
       returnV
     }
}
```

Listing B.7: Core of the unapply method of local synchronous channel

The above code implements the core algorithm and could be improved for better efficiency. Firstly, if the value of a message has been proved not to trigger the join pattern, the *matched* method invoked by the span iteration does not need to run complex test for that value. To this end, a HashSet *checkedMsg* could be introduced to record checked message values. The set should be cleared after the span iteration. Secondly, when a message is selected, popping it to the head of the message queue will save the later work of removing that message from the queue. Lastly, each channel that triggers a pattern only needs to be added

to the *nameset* once. Although inserting an element to a hashset is relatively cheap, the cost could be further reduced to the cost of comparing two integers. The full implementation for the unapply methods of synchronous channel is given at Listing B.8. The first case statement is used for improving the efficiency of code involving singleton pattern, where a pattern only contains one channel. More explanation for this decision will be given in §B.2.3.

```
def unapply(attr:Any) : Option[Arg]= attr match {
  case (ch:NameBase, arg:Any) => {// For singleton patterns
    if(ch == this){ Some(arg.asInstanceOf[(Int,Arg)]._2))
    }else{ None }
  }
  case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
   fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase) => {
    if (this == banedName) {return None}
    if(fixedMsg.contains(this)){
      Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2))
    }else{
      var checkedMsg = new HashSet[Arg]

      def matched(m:(Int,Arg)):Boolean = {
        if (checkedMsg(m._2)) {false} // the message has been checked
        else {
          checkedMsg += m._2
          (!(pattern.isDefinedAt(nameset, pattern,
                                 fixedMsg+((this, m)), dp+1, this))
          && (pattern.isDefinedAt((nameset, pattern,
                         fixedMsg+((this, m)), dp+1, banedName))))
        }
      }

      var returnV:Option[Arg] = None
      argQ.span(m => !matched(m)) match {
         case (_, MutableList()) => { returnV = None }
         case (ums, ms) => {
           val arg = ms.head // the message could trigger a pattern
           argQ = (((ums.+=:( arg )) ++ ms.tail).toQueue)
           // pop this message to the head of message queue
           if(dp == 1) {nameset.add(this); pushMsgTag(arg)}
           returnV = Some(arg._2)
         }
      }
      checkedMsg.clear
      returnV
    }
  }
}
```

Listing B.8: The unapply method of local synchronous channel

The *Join* class and the *and* object As said in the earlier section, join patterns are represented as a partial function in this library. An instance of the Join class is responsible for storing the join definition and attempting to fire a pattern on request. If the requested channel message association could fire a pattern, all channels involved in that pattern will be asked to remove the matched message; otherwise, the channel will be notified to pend the message to its message queue.

Although this library encourages using join patterns as a convenience constructor to synchronizing resources, actor model is popular at the time of implementing this library and not all channels need to be synchronized with others. For this reason, this library gives singleton patterns the privilege on pattern examination. Readers may wonder to what extent the efficiency will be affected by the above decision. To answer this question, consider a typical join definition where p patterns are defined and there are m channels on each pattern. At the time of a new message's arrival, there are n messages pending on each channel on average. On average, this library needs $O(p)$ time to check all patterns as if they are singleton patterns before spending $O(pmn)$ time checking all patterns as join patterns. Therefore, the additional checking will not significantly increase the cost of checking join patterns but will benefit programs that use singleton patterns.

```
1 class Join {
2  private var hasDefined = false
3  implicit val joinsOwner = this
4  private var joinPat: PartialFunction[Any, Any] = _
5
6  def join(joinPat: PartialFunction[Any, Any]) {
7   if(!hasDefined){
8     this.joinPat = joinPat
9     hasDefined = true
10   }else{
11     throw new Exception("Join definition has been set for"+this)
12   }
13  }
14
15  def trymatch(ch:NameBase, arg:Any) = synchronized {
16    var names: Set[NameBase] = new HashSet
17    try{
18      if(ch.isInstanceOf[SynName[Any, Any]])
          {ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)}
19      if(joinPat.isDefinedAt((ch, arg))){// optimization for singleton
          pattern
20        joinPat((ch,arg))
```

```
21      }else{
22        if(ch.isInstanceOf[SynName[Any, Any]]){
23          joinPat((names, this.joinPat, (new HashMap[NameBase,
                Any]+((ch, arg))), 1, new SynName))
24          ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)
25        }else{
26          joinPat((names, this.joinPat, (new HashMap[NameBase,
                Any]+((ch, arg))), 1, new AsyName))
27        }
28        names.foreach(n => {
29          if(n != ch) n.popArg
30        })
31      }
32    }catch{
33      case e:MatchError => {// no pattern is matched
34        if(ch.isInstanceOf[SynName[Any, Any]])
              {ch.asInstanceOf[SynName[Any,Any]].popMsgTag}
35        ch.pendArg(arg)
36      }
37    }
38  }
39}
```

Listing B.9: Code defines the Join class

The last thing is to define an *and* constructor which combines two or more channels in a join pattern. Indeed, this is surprisingly simple to some extent. Thanks to the syntactic sugar provided by Scala, the infix *and* operator in this library is defined as a binary operator that passes the same argument to both operands.

```
1 object and{
2   def unapply(attr:Any) = {
3       Some(attr,attr)
4   }
5 }
```

Listing B.10: Code defines the and object

B.2.4 Implementing distributed join calculus

The **DisJoin** class extends both the **Join** class, which supports join definitions, and the **Actor** trait, which enables distributed communication. In addition, the distributed join definition manages a name server which maps strings to its channels. Compared to a local join definition, a distributed join definition has two additional tasks: checking if distributed channels used at a remote sites are annotated with correct types and listening messages sending to distributed

channels. The code of the **DisJoin** class is presented in Listing *B*.11.

```
class DisJoin(port:Int, name: Symbol) extends Join with Actor{
 var channelMap = new HashMap[String, NameBase] //work as name server

 def registerChannel(name:String, ch:NameBase){
   assert(!channelMap.contains(name), name+" has been registered.")
   channelMap += ((name, ch))
 }

 def act(){
   RemoteActor.classLoader = getClass().getClassLoader()
   alive(port)
   register(name, self)

   loop(
     react{
       case JoinMessage(name, arg:Any) => {
         if (channelMap.contains(name)) {
           channelMap(name) match {
             case n : SynName[Any, Any] => sender ! n(arg)
             case n : AsyName[Any] => n(arg)
        }}}

       case SynNameCheck(name, argT, resT) => {
         if (channelMap.contains(name)) {
           sender ! (channelMap(name).argTypeEqual((argT,resT)))
         }else{
           sender ! NameNotFound
         }
       }
       case AsyNameCheck(name, argT) => {
         if (channelMap.contains(name)) {
           sender ! (channelMap(name).argTypeEqual(argT))
         }else{
           sender ! NameNotFound
      }}}
   )
}}
```

Listing B.11: Code defines the DisJoin Class

In this library, a distributed channel is indeed a stub of a remote local channel. When a distributed channel is initialized, its signature is checked at the place where its referring local channel is defined. Later, when a message is sent through this distributed channel, the message and the channel name is forwarded to the remote join definition where the referring local channel is defined. Consistent with the semantic of distributed join calculus, reduction,

if any, is performed at the location where the join pattern is defined. If the channel is a distributed synchronous channel, a reply value will be sent back to the remote caller. Listing B.12 illustrates how distributed synchronous channel is implemented. Distributed asynchronous channel is implemented in a similar way.

```
class DisSynName[Arg:Manifest, R:Manifest](n:String,
    owner:scala.actors.AbstractActor){
  val argT = implicitly[ClassManifest[Arg]]//type of arguments
  val resT = implicitly[ClassManifest[R]]//type of return value

  initial()// type checking etc.

  def apply(arg:Arg) :R = synchronized {
    (owner !? JoinMessage(n, arg)).asInstanceOf[R]
  }

  //check type etc.
  def initial() = synchronized {
    (owner !? SynNameCheck(n, argT, resT)) match {
      case true => Unit
      case false => throw new Error("Distributed channel initial
          error:"+
                                    "Channel " + n + " does not have
                                        type "+
                                    argT+ " => "+resT+".")
      case NameNotFound => throw new Error("name "+n+" is not found at
          "+owner)
}}}
```

Listing B.12: Code defines distributed synchronous channel

Lastly, the library also provides a function that simplifies the work of connection to a distributed join definition.

```
object DisJoin {
  def connect(addr:String, port:Int, name:Symbol):AbstractActor = {
    val peer = Node(addr, port)//location of the server
    RemoteActor.select(peer, name)
  }
}
```

B.3 Limitations and Future Improvements

B.3.1 Assumption on linear pattern

As with most of implementations that support join patterns, this library assumes that channels in each join pattern are pairwise distinct. Nevertheless, the current prototype implementation does not check the linear assumption for better simplicity.

Under the current implementation, a non-linear pattern

- will never be triggered if the channel involves a non-linear channel that takes two or more different messages. For example, the pattern $\{case\ c(1)\ and\ c(2) \Rightarrow println(3)\}$ will never fire.
- will work as a linear pattern if the all occurrences of a non-linear channel could take the same message. In this case, one or more variable names could be used to indicate the same message value. For example, $\{case\ c(m)\ and\ c(n) \Rightarrow println(m + n)\}$ will print 4 when c(2) is called.

B.3.2 Limited number of patterns in a single join definition

Due to the limitation of the current Scala compiler (version 2.9.1), the library also has an upper limit for the number of patterns and the number of channels in each pattern. Although the pattern of this limitation is not clear, the writer observed that a "sorted" join-definition may support more patterns.

For example,

```
1 case A(1) and B(1) and C(1) => println(1)
2 case A(2) and B(2) and D(2) => println(2)
```

is a "better" join-definition than

```
1 case A(1) and B(1) and C(1) => println(1)
2 case D(2) and B(2) and A(2) => println(2)
```

The compiler error: "java.lang.OutOfMemoryError: Java heap space" usually indicates the above limitation.

B.3.3 Unnatural usages of synchronous channels

Users of the current library may define a join-definition as follows

```
// bad join definition
object myjoin extends Join{
 object A extends AsyName[Int]
 object S extends SynName[Int, Int]
 join {
   case A(1) => S reply 2
   case S(n) => println("Hello World")
 }
}
```

In addition to the linear assumption, the current library further assumes that:

(i) the action part only reply values to synchronous channels that appeared in the left hand side of $\Rightarrow$. For this reason, the first pattern in the above example is invalid.

(ii) all synchronous channels in a pattern, if any, will receive one and only one value when the pattern fires. For this reason, the second pattern in the above example is invalid.

For assumption (i), the writer assumes that a program only needs to send a reply value to a synchronous channel on request. For assumption (ii), we think that all invocations on synchronous channels are expecting a reply value. Unlike some other libraries such as Cω. this library permits multiple synchronous channels in a single pattern.

Violating any of the above assumptions may be accepted by the system but usually causes deadlock or unexpected behaviour at run time.

B.3.4 Straightforward implementation for synchronous channels

Readers may have noticed that the implementation for synchronous channels are implemented according to its straightforward meaning rather than its formal definition in the join-calculus, which translates synchronous channels to asynchronous channels.

Admittedly, the translation in the join-calculus is a clever strategy to mimic the straightforward meaning of synchronous channels with less constructs. As the Scala programming language provides low-level concurrency constructs,

we think that a direct implementation for the synchronous channel is easier than and could be consistent with the indirect translation.

B.3.5 Type of the join pattern and the unapply methods

As an implementation in a static typed language, users would expect a clear type for the join pattern and the *unapply* methods in **AsyName**, **SynName**, and the *and* object. If the join pattern has type **T** ⇒ **Unit**, then the *unapply* methods in **AsyName** and **SynName** should have type **T** ⇒ **Option**[**Arg**], and the *unapply* method in the *and* object should have type **T** ⇒ **Option**[(**T**, **T**)]. The unusual implementation that passes partial function to the unapply methods indicates that **T** is a recursive type. Furthermore, due to the optimization for singleton patterns, **T** is also a Either type.

For earnest readers, **T** is ∀**T**.**Either**[(**NameBase**, **Any**), (**HashSet**[**NameBase**], **PartialFunction**[**T**, **Unit**], **HashMap**[**NameBase**, **Any**], **Int**, **NameBase**)]. Defining such a complex data type in a separate place may not be more helpful for readers than typing all parameters in each case statement. Moreover, general users do not need to understand this complex type to use this library. For above reasons, we simply replace **T** with the **Any** Type and manually verify type correctness of our implementation.

Bibliography

Agha, G. A. (1985). Actors: a model of concurrent computation in distributed systems. `http://dspace.mit.edu/handle/1721.1/6952`. MIT: AI Technical Reports (1964 - 2004).

Allen, R., Lo, N., and Brown, S. (2008). *Zend Framework in Action*. Manning Publications Co., Greenwich, CT, USA.

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:Model-View-Controller. `https://developer.apple.com/library/mac/documentation/general/conceptual/CocoaEncyclopedia/CocoaEncyclopedia.pdf`. [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. `http://dspace.mit.edu/handle/1721.1/41962`. MIT: AI Memos (1959 - 2004).

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 Da-Capo 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*.

Burbeck, S. (1987). Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`. Accessed on May 2013.

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

Cardelli, L. (1995). Obliq: A Language with Distributed Scope. *Computing Systems*, 8:27–59.

Cardelli, L. and Gordon, A. D. (1998). Mobile ambients. In *In Proceedings of POPL'98*. ACM Press.

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

Clinger, W. D. (1981). Foundations of Actor Semantics. `http://dspace.mit.edu/handle/1721.1/6935`. MIT: AI Technical Reports (1964 - 2004).

Cruz, L. R. G. and Aguirre, O. O. (2005). A virtual machine for the ambient calculus. In *Electrical and Electronics Engineering, 2005 2nd International Conference on*, pages 56–59.

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 non-parametric 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*.

Emir, B., Odersky, M., and Williams, J. (2007). Matching objects with patterns. In *ECOOP 2007 Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer.

Endrikat, S., Hanenberg, S., Robbes, R., and Stefik, A. (2014). How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA. ACM.

EPFL (2014). Scala online documentation. http://www.scala-lang.org/documentation/. Accessed on May 2014.

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., Fessant, F. L., Maranget, L., and Schmitt, A. (2003). Jocaml: A language for concurrent distributed and mobile programming. In *In Advanced Functional Programming*, pages 129–158. Springer Verlag.

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.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D. (1996). A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 406–421, London, UK. Springer-Verlag.

Fournet, C., Gonthier, G., and Rocquencourt, I. (1995). The reflexive cham and the join-calculus. In *In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

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. `http://dspace.mit.edu/handle/1721.1/57710`. MIT: AI Working Papers (1971 - 1995).

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.

Haller, P. and Van Cutsem, T. (2008). Implementing joins using extensible pattern matching. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg. Springer-Verlag.

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

HE, J. (2014a). TAkka. `https://github.com/Jiansen/TAkka`.

HE, J. (2014b). TAkka Play. `https://github.com/Jiansen/TAkka-Play`. Accessed on May 2014.

He, J., Wadler, P., and Trinder, P. (2014). Typecasting Actors: from Akka to Takka. *Scala 2014*.

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. *MIT: AI Memos (1959 - 2004)*.

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.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface*. Newnes.

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. `https://www.duo.uio.no/handle/10852/9621`.

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/. Accessed on May 2013.

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

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

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

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. C. (1997). A note on distributed computing. In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64, London, UK. Springer-Verlag.

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. http://en.wikipedia.org/wiki/Dining_philosophers_problem. [Online; accessed 12-November-2013].

Wikipedia (2013b). Mandelbrot set. http://en.wikipedia.org/wiki/Mandelbrot_set. [Online; accessed 12-November-2013].

Wikipedia (2013c). Sleeping barber problem. http://en.wikipedia.org/wiki/Sleeping_barber_problem. [Online; accessed 12-November-2013].

Wikipedia (2013d). Tic-tac-toe. http://en.wikipedia.org/wiki/Tic-tac-toe. [Online; accessed 12-November-2013].

Wikipedia (2014). Eight queens puzzle. http://en.wikipedia.org/wiki/Eight_queens_puzzle. [Online; accessed 30-March-2014].

WorldWide Conferencing, LLC (2013). Lift. http://liftweb.net/. Accessed on May 2013.

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.