Chapter 1 Introduction

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

A distributed application typically consists of components which handle some tasks independently, while collaborating on ~~some~~ other ~~task~~s by ~~sending~~ exchanging messages ~~to each other~~. 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.

Comment [TW1]: I added the comma to make completely clear that 'logic' only refers to the first half.

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, and formed in a tree structure, where a parent node is responsible for monitoring its children and restarting them when necessary. The supervision principle is proposed to increase the robustness of applications written in Erlang, a dynamically typed programming language. Erlang application developers can employ the supervision principle by using related APIs from the Erlang/OTP library. It is reported that the supervision principle helped AXD301, an ATM (Asynchronous Transfer Mode) switch manufactured by Ericsson Telecom AB. for British Telecom, to achieve 99.9999999% (9 nines) uptime during a nine-month test [Armstrong, 2002]. Nevertheless, adopting the Supervision principle is optional in Erlang applications.

Aside from employing good design patterns, programmers can use typed programming languages to construct reliable and maintainable programs. Typed programming languages have the advantages of detecting some errors earlier, enforcing disciplined and modular programming, providing a ~~safety~~ guarantee on language safety, efficiency optimisation, and other benefits [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~~s~~ the Dialyzer [Ericsson AB., 2013a] and a fault tolerance analysing 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 analysing tools is often involved in the later stages of Erlang applications development. In comparison with static analysing tools, simplified Erlang variants that use static type systems have been designed by Marlow and Wadler [1997], Sabelfeld and Mantel [2002], among others. As the expressiveness is often sacrificed in those simplified variants to some extent, code modifications are more or less required to make existing Erlang programs go through the type checker.

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

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

Comment [TW2]: This is currently using the slightly more Americanised 'z', but I've left them all in as I think it's accepted usage.

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

1. A typed name server is required to retrieve actor references of specific types. A distributed system usually requires a name server which maps names of services to processes that implement

that service. If processes are dynamically typed, the name server is usually implemented as a map from names to processes. Can a distributed name server maintain maps from the typed names and processes of corresponding types, and provide APIs for registering and fetching statically typed processes?

2. A supervisor actor must interact with child actors of different types. 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 ~~purpose~~ and ~~messages for~~ 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?

Comment [TW3]: Please confirm I've understood this correctly; the actor is both supervising nodes below, and doing its own processing, hence it needs to handle messages going to its children / parent, as well as deal with its own.

3. Actors that receive messages from distinct parties may suffer from the type pollution problem, ~~in which case~~whereby a party imports too much type information about an actor and can send the actor unexpected 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-parameterised actor publish itself as different types when it communicates with different parties?

Contributions

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

- The design and implementation of the TAkka library, where supervised actors are parameterized by the type of messages they expect. The library (Chapter **Error! Reference source not found.**~~Error: Reference source not found~~) mixes static~~al~~ and dynamic~~al~~ 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 ~~purpose~~ from those for actor specific communications. The decision is made so that type-parameterized actors of different types can form a supervision tree. In addition, the TAkka library is carefully

Comment [TW4]: Please confirm - I don't think the -al's were correct.

designed so that Akka programs can gradually migrate to their TAkka equivalents (evolution) rather than requiringe~~e~~ the provision ofding~~ding~~ type parameters everywhere (revolution). Moreover, the type pollution problem can be straightforwardly avoided in TAkka.

- A framework for evaluating libraries that support the supervision principle. The evaluation (Chapter **Error! Reference source not found.**~~Error: Reference source not found~~) compares the TAkka library and the Akka library in terms of expressiveness, efficiency and scalability. Results show that TAkka applications add minimal runtime overhead to the underlying Akka system and have a similar code size and scalability compared with their Akka equivalents. Finally, we port the Chaos Monkey library and design a Supervision View library. The Chaos Monkey library tests whether exceptions are properly handled by supervisors. The Supervision View library dynamically captures the structure of supervision trees. We believe that similar evaluations can be done in Erlang and new libraries that support the supervision principle.

Comment [TW5]: Please confirm, this reads that the the libraries are doing the supporting, not the framework (which would make it 'supports'.

Chapter 2 Background and Related Work

This Chapter summarises work that influences the design and implementation of the TAkka library. ~~This Chapter~~It begins with a general introduction on the Actor programming model and the Supervision principle~~. This Chapter ~~, then explains how to use the Actor programming model and the Supervision principle in the Erlang programming language and the Akka library. ~~This ~~The Chapter concl~~mplet~~es with a summary of the type system and pattern matching in Scala. The Actor model makes concurrent programming easy. The Supervision principle makes applications robust. The Supervision principle is introduced ~~by ~~in the Erlang language~~. It~~, and becomes obligatory in the Akka library, which is implemented in the Scala language. ~~The ~~Scala ~~language ~~has a sophisticated type system, which ~~enables ~~enabled the experimentals~~ of~~ building of ~~a ~~the more powerful and easier~~-~~to~~-~~use library, TAkka.

Comment [TW6]: Please confirm I've understood. The principle was first introduced in Erlang, and then when Akka arrived, it was made obligatory within Akka, not Erlang.

Comment [TW7]: Have I understood this, or is it meant to say it enables building of something(?) using the more powerful and easier library?

2.1 The Actor Programming Model

The Actor Programming Model is first proposed by Hewitt et al. [1973] for the purpose of constructing concurrent systems. In the model, a concurrent system consists of actors which are primitive computational components. Actors communicate with each other by sending messages. Each actor independently reacts to messages it receives.

The Actor model given in [Hewitt et al., 1973] does not specify its formal semantics and hence does not suggest implementation strategies ~~n~~either. An operational semantics of the Actor model is developed by Gerif [Grief, 1975]. Baker and Hewitt [1977] later define a set of axiomatic laws for Actor systems. Other semantics of the Actor model include~~s~~ the denotational semantics given by Clinger [1981] and the transition-based semantic model by Agha [1985]. Meanwhile, the Actor model has been implemented in Act 1 [Lieberman, 1981], a prototype programming language. The model influences designs of Concurrency Oriented Programming Languages (COPLs), especially the Erlang programming language [Armstrong, 2007b], which has been used in enterprise-level applications since it was developed in 1986.

Comment [TW8]: Please confirm which is correct

A recent trend is adding actor libraries to full-fledged popular programming languages that do not have actors built-in. Some of the recent actor libraries are JActor [JActor Consulting Ltd, 2013] for the JAVA language, Scala Actor [Haller and Odersky, 2006, 2007] for

Scala, Akka [Typesafe Inc. (b), 2012] for Java and Scala, and CloudHaskell [Epstein et al., 2011] for Haskell.

2.2 The Supervision Principle

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

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

The Akka library makes supervision obligatory. In Akka, every user-created actor is either a child of the system guidance actor or a child of another user-created actor. Therefore, every Akka actor is potentially the supervisor of some other actors. Unlike in the Erlang system, an Akka actor can be both a worker and a supervisor.

2.3 Erlang and OTP Design Principles

Erlang [Armstrong, 2007a, 2007b] 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~~g,~~ collaborates with other languages~~,~~ and provides fault-tolerant support for enterprise-level distributed real-time applications. One of the early OTP applications, Ericsson'~~â€™~~s AXD 301 switch, is reported to have achieved nine 9s availability, that is, 99.9999999% of uptime, during ~~the~~ 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).

Comment [TW9]: This should have an apostrophe before the 's' - the pdf doesn't show it but the rtf had some random characters instead!

Based on 10 years of experience ~~of~~ using Erlang in Enterprise level applications, Erlang developers summarized 5 OTP design principles in

1999 to improve the reliability of Erlang applications [Ericsson AB., 2013c]: The Behaviour Principle, The Application Principle, The Release Principle, The Release Handling Principle, and The Supervision Principle. The Supervision Principle ~~has been~~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 ~~the~~ object~~ed~~ oriented programming. It implements common structures and patterns of process implementations. With the help of behaviours, Erlang code can be divided into a generic part, a behaviour module, and a specific part, a callback module. Most processes, including the supervisor in Section **Error! Reference source not found.**~~Error: Reference source not found~~, ~~is~~ are coded by implementing a set of pre-defined callback functions for one or more behaviours. Although ad-hoc code and programming structures may be more efficient, using consistent general interfaces makes code more maintainable and reliable. Standard Erlang/OTP behaviours include:

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

Comment [TW10]: Please confirm

Comment [TW11]: slightly repetitious use of 'implements' and 'implementations' but might be unavoidable?

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~~s~~ two callback functions of the `application` behaviour: `start/2` and `stop/1`. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process, registered as `application_controller`.

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

Comment [TW12]: Do you mean 'at' rather than 'in'?

Comment [TW13]: likewise

2.3.3 The Release Principle and The Release Handling Principle

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

2.3.4 Applying OTP Design Principles in Java and Scala

To sum up, we make an analogy between Erlang/OTP design principles and common practices in Java and Scala programming, summarised in Table **Error! Reference source not found.**~~Error: Reference source not found~~.

OTP Design Principle	Java/Scala Analogy
Behaviour	defining an abstract class, an interface, or a trait.
Application	defining an abstract class that has two abstract methods: start and stop
Release	packaging related application classes
Release Handling	hot swapping support on key modules is required
Supervision	no direct correspondence

Table 1: Using OTP Design Principles in JAVA and Scala Programming

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

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

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

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

2.4 The Akka Library

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

2.4.1 Actor Programming in Akka

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

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 1: Akka Example: A String Counter

The code presented in Figure **Error! Reference source not found.**~~Error: Reference source not found~~ defines and uses an actor which counts String messages it receives. An Akka actor implements its message handler by defining a `receive` method of type `PartialFunction[Any, Unit]`. In Scala, `Any` is the supertype of all types. The type `Unit` has a unique value. A method with the return type `Unit`, such as the `receive` method, represents a block of local actions. An ~~analogous~~ analogy to such a method is a Java method which is declared `void`. In the `StringCounterTest` application, we create an Actor System (Section **Error! Reference source not found.**~~Error: Reference source not found~~), initialise an actor (Section **Error! Reference source not found.**~~Error: Reference source not found~~) inside the Actor System by passing a corresponding `Props` (Section **Error! Reference source not found.**~~Error: Reference source not found~~), and send messages to the created actor via its actor references (Section **Error! Reference source not found.**~~Error: Reference source not found~~). Unexpected messages to the `counter` actor (e.g. line 28 and 31) are handled by an instance of `MessageHandler`, a helper actor for the test application. Lastly, the order in which the four output messages are printed is non-deterministic, but "Hello World" is always printed before "Hello World Again" and "unhandled message:1" is always printed before "unhandled message:2".

2.4.1.1 Actor System

In Akka, every actor is resident in an Actor System. An actor system organises related actors in a tree structure and provides services such as thread scheduling, network connection, and logging. One or several local and remote actor systems ~~consist~~ 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 **Error! Reference source not found.**~~Error: Reference source not found~~ by the following code.

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

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

2.4.1.2 The Actor Class

An Akka Actor has four groups of fields given in Figure **Error! Reference source not found.**~~Error: Reference source not found~~: [*i*)] 1. its *state*, 2. its *behaviour* functions, 3. an `ActorContext` instance encapsulating its contextual information, and 4. 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.

Comment [TW14]: Sorry, I don't understand.

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

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

Comment [TW15]: Doesn't sound correct; they believe it 'should'? Or is 'believe' wrong?

There are two drawbacks to~~of~~ using mutable internal variables to represent states. Firstly, those variables will be reset each time when the actor is restarted, either due to a failure caused by itself or by~~e~~ enforce~~d~~ment by its supervisor for other reasons. Secondly, mutable internal variables result in the difficulty of implementing a consistent cluster environment where actors may be replicated to increase reliability [Kuhn et al., 2012]. The alternatives of working with mutable states will be discussed in Section **Error! Reference source not found.**~~Error: Reference source not found~~.

Comment [TW16]: Does this work?

There are two kinds of `behaviour` functions of an actor. The first type of behaviour function~~s~~ 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 implementations of th~~o~~ese four functions take no action when they are invoked.

Upon close inspection,~~Look closely at~~ it can be seen that the `receive` function of the `StringCounter` actor in Figure **Error! Reference source not found.**~~Error: Reference source not found~~, ~~it~~ actually has type `Function[String, Unit]` rather than the declared type `PartialFunction[Any, Unit]`. The definition of `StringCounter` is accepted by the Scala compiler because `PartialFunction` does not check the completeness of the input patterns. The behaviour of processing non-String messages, however, is undefined in the `receive` method.

2.4.1.3 Message Mailbox

An actor receives messages from other parts of the application. ~~Arrived~~ Arriving messages are queued in its sole mailbox to be processed. Differently ~~from~~ to the Erlang design (Section **Error! Reference source not found.**~~Error: Reference source not found~~), 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. This~~It~~ means that sending an ill-typed message will cause a failure at the receiver ~~side~~end. 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 perhaps stands for "properties", specifies the configuration used in~~of~~ creating an actor. A `Props` instance is immutable so that it can be consistently shared between threads and distributed nodes.

Comment [TW17]: I don't think you'll get away with uncertainty here!

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 3: Akka API: Props

We have seen an example of creating a `Props` instance in Figure **Error! Reference source not found.**~~Error: Reference source not found~~, that is:

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

which is short for

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

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

Comment [TW18]: singular or plural?

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:

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

In the above, the predefined function `classOf[T]` returns a class object for type `T`. More arguments can be sent to the constructor of `StringCounter` if there is one that requires more parameters. The signature of the constructor, including the number, types and order of its parameters, is verified at ~~the~~ run time. If no matched constructor is found when initializing the `Props` object, an `IllegalArgumentException` will arise.

Once an instance of `Props` is created, an actor can be created by passing that `Props` instance to the `actorOf` method of `ActorSystem` or `ActorContext`. In Figure **Error! Reference source not found.**~~Error: Reference source not found~~, we have seen that `system.actorOf` creates an actor directly supervised by the system guidance actor for all user-created actors (`user`). Calling `context.actorOf` creates an actor supervised by the actor represented by that context. Details of actor context and supervision will be given in Section **Error! Reference source not found.**~~Error: Reference source not found~~ and Section **Error! Reference source not found.**~~Error: Reference source not found~~ respectively.

2.4.1.5 Actor Reference and Actor Path

Actors collaborate by sending messages to each other via actor references of message receivers. An actor reference has type `ActorRef`, which provides a `!` method to which messages are sent. For example, in the `StringCounter` example in Figure **Error! Reference source not found.**~~Error: Reference source not found~~, `counter` is an actor reference to which the message `"Hello world"` is sent by the following code:

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

which is the syntactic sugar for

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 4: Akka API: Actor Reference

An actor path is a symbolic representation of the address where an actor can be located. Since actors forms a tree hierarchy in Akka, a unique address can be allocated for each actor by appending an actor name, which shall not ~~be~~ conflict with its siblings, to the address of its parent. Examples of Akka addresses are:

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

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

Comment [TW19]: you mean 'actor in question'?

An actor path corresponds to an address where an actor can be identified. It can be initialized without the creation of an actor. Moreover, an actor path can be re-used by a new actor after the termination of an old actor. Two actor paths are considered equivalent as long as their symbolic representations are equivalent strings. On the contrary, an actor reference must correspond to an existing actor, either an alive actor located at the corresponding actor path, or the special `DeadLetter` actor which receives messages sent to terminated actors. Two actor references are equivalent if they correspond to the same actor path and the same actor. 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`.

Comment [TW20]: You mean "Otherwise"?

2.4.1.6 Actor Context

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 5: Akka API: Actor Context

The API in Figure **Error! Reference source not found.**~~Error: Reference source not found~~ shows two groups of methods: those for interacting with other actors (lines 3 to ~~line~~ 24), and those for controlling the behaviour of the represented actor (lines 26 to ~~line~~ 30).

As mentioned in Section **Error! Reference source not found.**~~Error: Reference source not found~~, 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~~ed~~ actor, an `InvalidActorNameException` raises. If the user does not provide a name when creating an actor, a system generated name will be used instead. The return value of the `actorOf` method is an actor reference pointing to the created actor.

Once an actor is created, its actor reference can be obtained by inquiring on its actor path using the `actorFor` method. Since version 2.1, Akka encourages the obtaining of actor references via a new method `actorSelection`, whose return value broadcasts messages it receives to all actors in its subtrees. The `actorFor` method is deprecated in version 2.2. Code in this thesis still uses the deprecated `actorFor` method because, in most cases, our simple examples only need to send messages to a specific group of actor.

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

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~~ing~~ the lifecycle of an actor either inside the actor or via its supervisor, we believe that those three methods are redundant in Akka. For all examples studied in this thesis, there is no client application that requires any of those three methods.

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

2.4.1.7 Dynamic Behaviour Swapping

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 6: Akka Behaviour Swap Example

In the `StringCounter` example given at the beginning of this section, a message handler is defined in the `receive` method. The `StringCounter` is a simple actor which only requires an initial message handler that never changes. In some other cases, ~~it is required~~

~~to update~~ the message handler of an actor 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 **Error! Reference source not found.**~~Error: Reference source not found~~ defines a calculator whose behaviour changes at run-time. The calculator starts with a basic version that can only compute multiplication. When it receives an `Upgrade` command, it upgrades to an advanced version that can compute both multiplication and division. The advanced calculator downgrades to the basic version when it receives a `Downgrade` command. For simplicity, the demo code does not consider the potential *division by zero* problem, an error that can be tolerated if the actor is properly supervised.

2.4.2 Supervision in Akka

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

A distinguishing feature of the Akka library is making supervision obligatory by restricting the way ~~of creating~~ actors are created. Recall that every user-created actor is initialised in one of ~~the~~ two ways: using the `system.actorOf` method so that it is a child of the system guardian actor; or using the context.actorOf method so that it is a child of another user-created actor. Therefore, all user-created actors in an actor system, together with the guardian actor of that actor system, form a tree structure. Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance. This section summarises concepts in the Akka supervision tree.

2.4.2.1 Children

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

2.4.2.2 Supervisor Strategy

The Akka library implements two supervisor strategies: `OneForOne` and `AllForOne`. The `OneForOne` supervisor strategy corresponds to the `one_for_one` supervision strategy in OTP, which 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 an Akka actor does not specify the order of its children. Simulating the `rest_for_all` strategy in Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. It is not clear whether the lack of the `rest_for_one` strategy will result in difficulties when rewriting Erlang applications in Akka.

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 7: Akka API: Supervisor Strategies

Figure **Error! Reference source not found.**~~Error: Reference source not found~~ gives the API of 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.

Comment [TW21]: Please confirm 'the' (there is only one API possible) rather than 'an' (more than one is possible, we are showing one of them).

Comment [TW22]: should this be "for"?

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 8: Akka Example: Supervised Calculator

Figure **Error! Reference source not found.**~~Error: Reference source not found~~ defines a simple calculator which supports multiplication and division. The simple calculator does not consider the problematic case of dividing a number by 0, in which case an `ArithmeticException` will raise. We then define a safe calculator as the supervisor of the simple calculator. The safe calculator delegates calculation tasks to the simple calculator and restarts the simple calculator when an `ArithmeticException` is raiseds~~s~~. 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 messages ~~is~~ are delivered. The last message is not processed ~~because~~ since ~~the~~ both calculators are terminated because the simple calculator fails more frequently than allowed.

2.5 The Scala Type System

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

2.5.1 Parameterized Types

A *parameterized type* `T[U`$_1$`,…,U`$_n$`]` consists of a type constructor `T` and a positive number of type parameters `U`$_1$`,…,U`$_n$ [Odersky., 2013]. The type constructor `T` must be a valid type name whereas a type parameter `U`$_i$ can either be a specific type value or a type variable. Scala Parameterized Types are similar to Java and C# generics and C++ templates, but express *variance* and *bounds* differently as explained later.

2.5.1.1 Generic Programming

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 9: Scala Example: A Generic Stack Library

To demonstrate how to use Scala parameterized types to do generic programming, Figure **Error! Reference source not found.**~~Error: Reference source not found~~ gives a simple stack library and an associated client application ported from a Java example found in [Naftalin and Wadler, 2006, Example 5-2]. The example defines an abstract data type Stack, an implementation class `ArrayStack`, a utility method `reverse`, and client application `Client`.

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

The `ArrayStack` class implements the `Stack` trait and overrides the `toString` method which gives a string representation of the `Stack`. An `ArrayStack` instance internally saves data in a~~-~~ `List` so that both prepending an element and removing the first element take constant time.

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

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

Comment [TW23]: you mean a feature of the compiler, or that there is a compiler called autoboxing? Sorry I don't know enough so I wanted to check.

Comment [TW24]: You'll notice that in the previous paragraph or two I've shortened several double spaces into single ones - were they intentional? I'll leave them from now on but comment them.

2.5.1.2 Type Bounds

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

Figure **Error! Reference source not found.**~~Error: Reference source not found~~ gives a more flexible Stack, the types of whose elements are either the same as the type parameter or subtypes of the type parameter. In Figure **Error! Reference source not found.**~~Error: Reference source not found~~, the signature of `push` is changed to `push[T<:E](elt:T):Unit`, with an additional type parameter `T<:E` which denotes that `T` is a subtype of `E`. In Scala, `E` is called the *upper bound* of `T`. Similarly, `T>:E` means `T` is a supertype of `E` and `E` is called the *lower bound* of `T`. In Scala, `Any` is the supertype of all types and `Nothing` is the subtype of all types.

The remaining code in Figure **Error! Reference source not found.**~~Error: Reference source not found~~ is the same as the code in Figure **Error! Reference source not found.**~~Error: Reference source not found~~ except that, on line 4 of the `Client` example, the value of `i` needs to be explicitly converted to an `Integer`.

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 10: Scala Example: A Generic Stack Library using Type Bounds

2.5.1.3 Variance Under Inheritance

An important issue that is intentionally skirted in Section **Error! Reference source not found.**~~Error: Reference source not found~~ is how variance under inheritance works in Scala. Specifically, if T_{sub} is a subtype of `T`, is `Stack[`T_{sub}`]` the subtype of `Stack[T]`, or the reverse~~ly~~? Unlike Java generic collections [Naftalin and Wadler, 2006], which are always invariant on the type parameter, Scala users can explicitly specify one of the three types of variance as part of the type declaration using variance annotation as summarised in Table **Error! Reference source not found.**~~Error: Reference source not found~~, paraphrased from [Wampler and Payne, 2009, Table 12.1].

A variance annotation constraints positions where the annotated type variable may appear. Specifically, covariant, contravariant, and invariant type variables can only appear in covariant

Comment [TW25]: The rtf is unreadable here so I've copied the paragraph from the .pdf

Comment [TW26]: Should be singular? Or do you mean 'constrains' - which I think?

position, contravariant position, and invariant position respectively. The Scala compiler checks if types with variance are used consistently according to a set of rules given in [Odersky., 2013, Section 4.5]. As a programmer, the author of this thesis often find that it is easier to uses variant types according to a variant of the Get and Put Principle.

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

Description	
+	Covariant subclassing. i.e. X[Tsub] is a subtype of X[T], if Tsub is a subtype of T.
-	Contravariant subclassing. i.e. X[T] is a subtype of X[T], if Tsup is a supertype of T.
default	Invariant subclassing. i.e. cannot substitute X[T] or X[Tsub] for X[T], if Tsub is a subtype of T and T is a subtype of Tsup.

Table 2: Variance Under Inheritance

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

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

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

Comment [TW27]: This is the original version, which is now legible since I pasted in the copy. I'll leave it in here.

Formatted: Highlight

Formatted: Highlight

When us~~e~~inge generic types with variance in Scala, the general version is:

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]
[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 11: 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 that is covariant on its type parameter, as shown in Figure **Error! Reference source not found.**~~Error: Reference source not found~~. A stack is covariant on its type parameter because, for example, a stack that saves a collection of `Integer` values is also a stack that saves a collection of `Any` values. However, if the type of Stack is declared as `Stack[+E]`, the signature of its `push` method *cannot* be

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

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

2.5.2 Scala Type Descriptors

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

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

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 12: Scala API: Manifest Type Hierarchy

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 13: Scala ExampleI: Manifest Example

The code example in Figure **Error! Reference source not found.**~~Error: Reference source not found~~ shows common usages of Manifest. There are three ways of obtaining a manifest: using the Methods `manifest` (line 16) or `classManifest` (line 12), using an implicit parameter of type `Manifest[T]` (line 21), or using a `context bound` of a type parameter (line 26). Context bound can be seen as a syntactic sugar for implicit parameters without a user-specified parameter name. The `isSubType` method defined at line 31 tests if the first `Manifest` represents a type that is a subtype of the typed represented by the second `Manifest`.

Comment [TW28]: multiple spaces

Comment [TW29]: double space

2.6 Function and Partial Function

[Sorry. Ignored `\begin{lstlisting} ... \end{lstlisting}`]

Figure 14: Scala Example: PartialFunction and Function

There are two distinctions between `Function` and `PartialFunction`. The advantage of `PartialFunction` is that users can define a new function that merges the input domains of two other partial functions. The advantage of `Function` is that the completeness of pattern matching can be checked by the compiler if the input type of the function is a sealed trait or a sealed class, whose sub-classes must be defined in the same file. Figure **Error! Reference source not found.**~~Error: Reference source not found~~ gives a Scala example that illustrates the above features. The last function, `fruitNameF`, shows that the syntactically shorter definition of `fruitnamePF` can be equivalently defined. Moreover, because the `typedReceive` function should not be visible outside the Actor class, the advantage of using `PartialFunction` is not clear. On the other hand, a completeness check of message patterns might be a useful feature in practice.

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 1999 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 ~~that~~ using the supervision principle have achieved ~~a~~ high availability [Armstrong, 2002]. Recently, the actor programming model and the supervision principle have been ported to Akka, an Actor library written in Scala. Although Scala is a statically typed language and provides a sophistical type system, the type of messages sent to Akka actors are dynamically checked when they are processed. The next chapter presents the design and implementation of the TAkka library where type checks are involved ~~in~~ at the earliest opportunity to expose type errors.

Comment [TW30]: sophisticated? I know this is a real word but am not sure if you meant it?