~~Chatpet~~ Chapter 5

TAkka: Evaluation

This section evaluates the TAkka library with regard~~sing~~ to the following three aspects. Firstly, Section X shows that the notion of type-parameterized actor in the TAkka library has syntactical advantages to avoid ~~the~~ Wadler's type pollution problem. Secondly, Sections X and X show that rewriting Akka programs using TAkka will *not* bring~~ing~~ obvious runtime and code-size overheads. Thirdly, Section X gives two accessory libraries, ChaosMonkey and SupervisionView, for testing the reliability and availability of TAkka applications.

5.1 Wadler's Type Pollution Problem

~~The~~ Wadler's type pollution problem refers to the situation where a communication interface of a component publishes too much type information to another party and consequently that party can send the component a message not expected from it. Without due care, actor-based systems that are constructed using ~~the~~ layered architecture [Dijkstra, 1968; Buschmann et al., 2007] or the MVC pattern [Reenskaug, 1979, 2003] can suffer from the type pollution problem.

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

The other solution is publishing a component as of different types to different parties. The published type shall be a supertype of the most precise type of the component. In Java and Scala applications, this solution can~~ould~~ be tricky because, as briefly discussed in Section X, 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 implemented using the MVC pattern.

5.1.1 Case Study: Tic-Tac-Toe

5.1.1.1 The Game

Tic-Tac-Toe [Wikipedia, 2013d], also known as Noughts and Crosses, is a paper-and-pencil game. A basic version of the Tic-Tac-Toe game is played by two players who mark X and O in turn in a 3×3 grid. A player wins the game if he 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 X gives an example game won by the first player, X. Figures X to X are screenshots of the game implemented in the next subsection. The graphical user interface of the game contains three parts. The left hand side of the window shows the player ~~of~~ with the next move. The middle of the window shows the current status of the game board. The right hand side contains control buttons. Finally, Figure X announces the winner .

Figure 1: A Game of Tic-Tac-Toe

5.1.1.2 The MVC pattern

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

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

situation where~~n~~ ~~a~~ the model sends~~t~~ the controller a message expected from the~~a~~ view, ~~and~~ or the ~~case that a~~ view pretends to be~~-~~ the~~a~~ model.

5.1.2 A TAkka Implementation

Code

Figure 2: TicTacToe: Message

Code

Figure 3: TicTacToe: Model

Code

Figure 4: TicTacToe: View

Code

Figure 5: TicTacToe: Application

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

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

In the definition of the `Model` actor (Figure X) and the `View` actor (Figure X), 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~~s~~ array. Each time the model receives a request from the controller, it updates the status of the board and then announce the winner or the next player to the controller. The `View` actor maintains a GUI application that displays the game board and listens to user input. All user input ~~are~~ 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.

Finally, setting up the application is straightforward. In the code given at the bottom of Figure X, 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 given~~s~~ in Figure X.

5.1.3 A Scala Interface

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

Similar to the TAkka implementation which separates the type of messages sent from a model to a controller and a view to a controller, the interface in Figure X 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 is added to ~~any~~ either of the two traits due to the changes of application requirement, the whole program needs to be recompiled and re-deployed. ~~In the case that~~ Where an application is a collaborative~~ed~~ project~~s~~ maintained by different groups, attempts at~~s~~ ~~of updating~~large-scale updates ~~such an application throughly shall~~should be avoided whenever possible.

~~On the~~In contrast, in our TAkka implementation, new messages can be added easily. With the benefit of backward compatible hot swapping

Comment [TW1]: Sorry, I don't understand on message ~~message~~ handlers, the controller, the model and the view can each be updated separately.

Code

Figure 6: TicTacToe: MVC Interface

5.2 Expressiveness and Correctness

5.2.1 Examples

To assess the expressiveness and correctness of the TAkka library, w~~.~~ ~~W~~e selected examples from Erlang Qu~~i~~viQ~~q~~ [Arts et al., ~~[~~2006] and open source Akka projects to ensure that the main requirements for actor programming we~~a~~re not unintentionally neglected. Examples from Erlang QuviQ ~~Quiviq~~ we~~a~~re re-implemented using both Akka and TAkka. Examples from Akka projects we~~a~~re re-implemented using TAkka.

5.2.1.1 Examples from the Quviq Project

QuviQ ~~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 ~~Quivq~~ is a commercial product. The author gratefully acknowledges Thomas Arts from Quvi~~iv~~q.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 versions ported from the Erlang source code. This thesis will only describe the overall structure of those two examples. The Akka and TAkka code is available in the ♠**private repository** ♠ but is not available in the ♠**public repository** ♠. Readers who would like ~~to have~~ access to the Erlang source code ~~shall~~ should contact Quvi~~iv~~q.com and Erlang Solutions directly.

ATM simulator

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

Comment [TW2]: You say Quivi

Figure 7: Example: ATM

Elevator Controller

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

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

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

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

Figure 8: Example: Elevator Controller

5.2.1.2 Examples from the Akka Documentation

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

Comment [TW3]: Please check I've corrected this properly

Ping Pong

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

Dining Philosophers

This example contains two actor classes. It models the dining philosophers problem [Wikipedia, 2013a] using Finite State Machines (FSMs). The Dining Philosophers problem [Wikipedia, 2013a] is one of the classical problems that exhibit~~s~~ synchronization issues in concurrent programming. In the ported version five philosophers sit around ~~at~~ a table with five chopsticks interleaved. Each philosopher alternately thinks and eats. Before starting~~ed~~ 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 four actor classes. It demonstrates distributed computation~~s~~ and hot code swapping on the receive function of an actor. The TAkka version of this example is used as a case study in Section X.

Fault Tolerance

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

5.2.1.3 Examples from Other Open Source Projects

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

Barber Shop

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

EnMAS

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

Socko Web Server

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

Gatling

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

Play Core

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

5.2.2 Evaluation Results

5.2.2.1 Code Size

Following the suggestion by Fleming and Wallace [1986] and Hennessy and Patterson [2006], we assess the overall code modification and code size by calculating the geometric mean of all examples. The evaluation results in Table X show that when porting an Akka program to TAkka, about 7.4% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because TAkka code does not need to handle unexpected messages. On average, the total program size of Akka and TAkka applications are almost the same.

5.2.2.2 Type Error

A type error is reported by the compiler when porting the Socko example [Imtarnasan and Bolton, 2012] from its Akka implementation to an equivalent TAkka implementation. SOCKO is a library for building event-driven web services. The SOCKO designer defines a `SockoEvent` class to be the supertype of all events. One subtype of

SockoEvent is HttpRequestEvent, representing events generated when an HTTP request is received. The designer further implements subclasses of Method, whose unapply method intends to pattern match SockoEvent to HttpRequestEvent. The SOCKO designer made a type error in the method declaration so that the unapply method pattern matches SockoEvent to SockoEvent. The type error is not exposed in test examples because ~~the~~ those examples always pass instances of HttpRequestEvent to the unapply method and send the returned values to an actor that accepts messages of HttpRequestEvent type. Fortunately, the design flaw is exposed when upgrading the SOCKO implementation using TAkka.

[Sorry. Ignored \begin{sidewaystable} ... \end{sidewaystable}]

5.3 Throughput

The Play example [Typesafe Inc. (c), 2013] and the Socko example [Imtarnasan and Bolton, 2012] used in Section X 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 ~~handled~~ 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 Soc~~co~~ko.

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

CODE

Figure 9: Example: JSON serialization Benchmark

All four versions of the web services we~~a~~re deployed to servers on Amazon Elastic Compute Cloud (EC2) [Amazon.com, Inc., 2013a]. The example ~~is~~ was tested with up to 16 EC2 micro instances (t1.micro), each of which had~~s~~ 0.615 GB Memory. We 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], we further designed and implemented the FreeBench [HE, 2013] tool, a benchmarking tool for HTTP servers. One feature of the FreeBench tool is that it can benchmark web servers deployed at multiple addresses. In the test, we confirmed that, for the JSON serialization example, the maximum throughput achieved when using the Elastic Load Balancing (ELB) service [Amazon.com, Inc., 2013b] wa~~i~~s not significantly improved when more servers we~~a~~re added. On the other hand, when we benchmarked all deployed EC2 servers, the total throughput achieved wa~~i~~s increased slightly. One possible explanation for the above observation is that, the load balancer used in the first test, which runs on a micro EC2 instance, is the I/O bound of the throughput measurement. Another feature of the FreeBench tool is that it can be configured to carry out a number of benchmarks in parallel and repeat the parallel benchmark for a certain number of times. The benchmark results of all tests are sent to a data store which reports a customised statistical summary.

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

Figure X summaries the result of the JSON serialization benchmark. It shows the average and the standard de~~ri~~viation of the throughput in each test. The result shows that web servers built using an Akka-based library and an TAkka-based library have similar throughput.

Figure 10: Throughput Benchmarks

5.4 Efficiency and Scalability

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

1. the cost of adding an additional operational layer on top of Akka code,
2. the cost of constructing type descriptors, and
3. the cost of transmitting type descriptors in distributed settings.
4. the cost of dynamic type checking when registering new typed names and hot swapping message handlers.

We assessed the upper bounds ~~of the cost~~ of costs i) and ii) by a micro benchmark which assessed~~s~~ the time for~~of~~ initializing *n* instances of `StringCounter` defined in Figure X and Figure X. When *n* ranges from ? to ?, as shown in Figure X, the TAkka implementation is about 2 times slower than~~as~~ the Akka implementation.

Comment [TW4]: This isn't very clear; do you mean 'runs roughly half as fast as'? That would be the clearest way I think.

Figure 11: Cost of Actor Construction

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

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

5.4.1 BenchErl Overview

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

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

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

5.4.2 Benchmark Examples

5.4.2.1 Ported BenchErl Examples

The following BenchErl examples we~~a~~re 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~~s~~ 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 I/O bounded when the number of child processes is large.

big

This benchmark tests many-to-many message passing. A child process in this example sends a `Ping` message to each of the other child processes. Meanwhile, each child replies 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 ~~completed~~ 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. ~~Same~~ Similarly to~~as~~ the `bang` example, the benchmark is likely to be I/O bounded when the number of child processes is large.

ehb

This benchmark re-implements the hackbench example [Zhang, 2008] originally used for stress testing ~~for~~ Linux schedulers. Each child process in this benchmark is a group of message senders and receivers. Each sender sends each receiver a dummy message and waits for an acknowledge message. Each sender repeats the process ~~for~~ a number of times. When a sender has received all expected replies, it reports to the child actor that it has completed~~s~~ its task. When all senders in the group have ~~completes~~ completed their tasks, the child process sends a complete message to the master process. The benchmark completes when all child processes have finished~~es~~ 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 2*m*2 messages are sent in each group. Therefore, the main task of this benchmark is sending messages inside each group. With the increased number of available nodes to share the task of child processes, this benchmark is expected to have shorter runtime.

Comment [TW5]:

Comment [TW6]: This was missing in the Word version and I've not superscripted the 'squared'.

Formatted: Font: Italic

genstress

This benchmark is similar to the bang test. It spawns an echo server and a number of clients. Each client sends some dummy messages to the server and waits for its response. When a client receives the response, it sends an acknowledge message to the master process. The benchmark completes when results from all child processes are received. The Erlang implementation has two versions, one using the OTP *gen_server* behaviour, the other implementing~~s~~ a simple server-client structure manually. For generality, this benchmark ports the version ~~without~~ not using *gen_server*.

Comment [TW7]: please check I've expressed this correctly. Perhaps best written "There are two versions in Erlang,"

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

mbrot

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

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

parallel

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

Parameters set in this example are the number of available nodes, the number of child processes, and the number of timestamps each child ~~to~~ 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 ~~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~~a~~ 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 I/O bounded when the speed of generating messages exceeds the speed of forwarding messages.

5.4.2.2 BenchErl Examples that are Not Ported

The following BenchErl examples are not ported for reasons given in respectiv~~ed~~e paragraphs.

ets_test

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

pcmark

~~Same as~~Similarly to the *ets_test* example, this benchmark also tests the ETS operations. In this benchmark, five ETS tables are created. Each created table is filled with some values before the benchmark begins. The benchmark spawns a certain number of child processes that read

Comment [TW8]: I think this should be removed but I don't know the standard usage.

the content of those tables. This example is not ported ~~as well~~either because it uses ETS table.

Comment [TW9]: 'the'? or 'an'?

timer_wheel

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

5.4.3 Benchmark Methodology

5.4.3.1 Testing Environment

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

Comment [TW10]: switches?

5.4.3.2 Determining Parameters

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

First, except for the RUN example, the runtime of each experiment wa~~i~~s ~~controlled~~ constrained to be under 40 seconds. The decision ~~is~~ was made so that the time to measure each example ~~is~~ was acceptable.

As we will explain in the next section, each example ~~are~~ was tested for a total of 90 rounds of experiments. Another reason for this decision wa~~i~~s that the experiment should be able to complete in a reasonably~~e~~ short time when running on a single machine. During the experiment, we fou~~i~~nd some configurations such that an example had~~s~~ a bad performance when run~~s~~ on a single machine but spe~~e~~d up significantly when another machine ~~shares~~ shared part of its workload. ~~Those~~ These experiments ~~give~~ gave interesting results that proved the importance of distributed programming~~,~~ ; however~~;~~ we ~~would like~~desired that the number of nodes be the only independent variable throughout all experiments. Therefore, we preferred ~~the~~ configurations that neglec~~t~~ted~~s~~ the impact of other factors such as garbage collection.

Second, we preferred configurations that ~~have~~ had more workload for each child process. In a number of tests to determine parameters, we observed that employing more ~~number of~~ machines only ~~have~~ had runtime benefit for those BenchErl examples whose runtime is bounded by the computational tasks rather than the I/O cost of communicating with the only master process. The only exception ~~of~~ to this principle ~~is~~ was the Parallel Fibonacci Numbers example for the verification of our observation.

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

Based on the results of our trial experiments, the parameters used in each example ~~are~~ were ~~the~~ as follow~~ing~~s:

bang

- number of child processes: 512
- number of messages sent by each child process: 2,000

big

- number of child processes: 1,024

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: 6,000 x 6,000

parallel

- number of child processes: 256
- number of timestamps each child to create: 6,000

ran

- number of child processes: 6,000
- list size: 100,000

serialmsg

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

5.4.3.3 Measurement Methodology

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

5.4.4 Evaluation Results

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

Comment [TW11]: singular or plural ("a different number")?

Observation 1

In all examples, TAkka and Akka implementations ~~have~~ had almost identical run-times and hence have similar scalability. In Figure X, the runtime of Akka benchmarks and TAkka benchmarks often overlay ~~on~~ each other. For benchmarks that do not overlay~~s~~, the difference is less than 10% on average. The scalability of Akka applications and the scalability of their TAkka equivalents appear~~s~~ slightly different because their differences ~~of them is~~are amplified by ~~the differences of~~ 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. ~~Those~~ These examples do not scale when the number of nodes are greater than a certain number. Examples of this observation are the `Bang` example, the `Big` example, and the `Run` example. The speed-up of those examples does not further increase when the number of nodes is more than four or eight.

Observation 4

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

Figure 12 – 14 : Runtime and Efficiency Benchmarks

Comment [TW12]:

Comment [TW13R12]:

Comment [TW14R13]: This looks wrong.

5.4.5 Additional Benchmark: Parallel Fibonacci Numbers

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

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

Equation (1)

The above basic way of computing a Fibonacci number ~~is~~ was chosen because it has a quadratic 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*). We expected that, when setting the number of child processes to a number that is higher than the number of available nodes, a benchmark with a higher *n* would give~~s~~ better scalability than those with a lower *n*.

♠redo the Fibonacci example on Beowulf ♠

[Fib10 Time (To Update)]

Figure 15: Benchmark: Parallel Fibonacci Numbers

5.5 Assessing System Reliability and Availability

The supervision tree principle is adopted by Erlang and Akka users ~~with~~ in the hope of improving the reliability of software applications. Apart from the 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 X) show that failures occur during the test. On the other hand, any failed actors are restored, and hence appropriately supervised applications (Section X) pass Chaos Monkey tests.

Code

Figure 16: TAkka Chaos Monkey

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

Comment [TW15]: nodes?

messages.

Table 1: TAkka Chaos Monkey Modes

5.5.1 Chaos Monkey and Supervision View

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

Figure X gives the API and the core ~~implemention~~implementation of TAkka Chaos Monkey. A user sets up a Chaos Monkey test by initializing a `ChaosMonkey` instance, defining the test mode, and scheduling the interval between each run. In each run, the `ChaosMonkey` instance sends a randomly picked actor a special message. ~~When~~ Upon receivin~~g~~e a Chaos Monkey message, a TAkka actor executes a corresponding potentially problematic code as described in Table X. `PoisonPill` and `Kill` are handled by `systemMessageHandler` and can be ~~overrided~~ overridden as described in Section X. `ChaosException` and `ChaosNonTerminate`, on the other hand, are handled by the TAkka library and cannot be overridden~~ed~~.

Comment [TW16]: a piece of code? or just 'code'?

5.5.2 Supervision View

To dynamically monitor changes of supervision trees, we designed and implemented a Supervision View library. In a supervision view test, an instance of `ViewMaster` periodically sends request messages to interested actors. ~~At the time w~~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 reply is sent. The `ViewMaster` instance records status messages and passes them to a visualizer, which will analyze and interpret changes of the tree structure during the testing period.

Comment [TW17]: in?

Comment [TW18]: in?

A view master is initialized by calling one of the apply methods of the `ViewMaster` object as given in Figure X. Each view master has an actor system and a master actor as its fields. The actor system is set up according to the given `name` and `config`, or the default configuration. The master actor, created in the actor system, has type `TypedActor[SupervisionViewMessage]`. After the start method of a view master is called, the view master periodically sends `SupervisionViewRequest` to interested nodes in supervision trees, where `date` is the system time just before the view master sends requests. When a TAkka actor receives a `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 the appropriate drawer on request.

Comment [TW19]: an?

CODE

Figure 17: Supervision View

Figure

Figure 18: Supervision View Example

5.5.3 A Partly Failed Safe Calculator

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

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

5.5.4 BenchErl Examples with Different Supervisor Strategies

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

5.6 Summing Up

This chapter confirms that TAkka can detect type errors without bringing in obvious overheads. 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 wa~~i~~s found in the Socko framework. We also show that TAkka has ~~its~~ the advantage ~~to~~ of ~~solve~~ solving the Wadler's type pollution problem. Secondly, web servers built using Akka and TAkka reach~~es~~ similar throughput when the same number of EC2 instances are used. Thirdly, BenchErl benchmark examples written in Akka and TAkka have ~~a~~ similar efficiency and scalability when running on a 32 node Beowulf cluster. The additional benchmark example in Section X ~~gives~~ provides ~~an~~ evidence that the scalability of an application depends on the ratio of the cost of parallelized computational tasks and the cost of I/O bounded communications. Lastly, we provides a ChaosMonkey library and a Supervision View library for assessing the correctness of applications built using TAkka.

Chapter 6

Future Work

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

6.1 Supervision and Typed Actor in Other Systems

The result of this thesis confirms the feasibility of using type parameterized actors in a supervision tree. The 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 allow us to explore types in a supervision tree. In TAkka, dynamic type checking is only used when static type checking meets its limitations. Thirdly, Akka is a popular programming framework. As part of the T~~t~~ypesafe stack, Akka has been used for developing applications in different sizes and for different purposes. If Akka applications can be gradually upgraded to TAkka applications, we believe that the type checking feature in TAkka can improve the reliability of existing Akka systems.

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

6.2 Benchmark Results from Large Real Applications

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

6.3 Supervision Tree that Supports Software Rejuvenation

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

6.4 Measuring and Predicting System Reliability

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

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

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

Comment [TW20]: Not sure if this needs 'a' before 'simple' in both occurrences, or 'the', or neither. It depends. Or 'nodes' could even be plural. Sorry I don't know the terminology.

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

Chapter 7

Thesis Summary

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

7.1 Overview of Contributions

7.1.1 A library for Type-parameterized Actors and Their Supervision

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

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

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

7.1.2 A Library Evaluation Framework

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

Compared with Akka, the TAkka library avoids the Wadler's type pollution problem straightforwardly. The type pollution problem, discussed in Section X, 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. Our demonstration example shows that avoiding Wadler's type pollution problem in TAkka is as simple as publishing a service as having different types when it is used by different parties.

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

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

7.2 Conclusion

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

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