Hot Code Reloading in Cloud Haskell

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Pankaj More

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2014

CERTIFICATE

It is certified that the work contained in the thesis titled **Hot Code Reloading in Cloud Haskell**, by **Pankaj More**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof Amey Karkare

Department of Computer Science & Engineering

IIT Kanpur

June, 2014

ABSTRACT

Name of student: **Pankaj More** Roll no: **Y9227402**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Hot Code Reloading in Cloud Haskell**

Name of Thesis Supervisor: **Prof Amey Karkare**

Month and year of thesis submission: **June, 2014**

Zero downtime is a fairly common requirement in large scale distributed systems running critical business logic. But, most of the software run in production today is not built with this requirement in mind. The standard approach to maintain up-time is by using redundant hardware which is both expensive and complex.

The ability of a program to update itself while in execution is commonly know as Dynamic Software Updating. DSU also helps in rapid prototyping. The ability to modify a running program on the fly is also useful in supporting interactive programming.

Cloud Haskell is a framework for writing distributed applications in Haskell similar to Erlang. In this thesis, we try to solve the problem of hot-code reloading a Cloud Haskell application. We first try to figure out Erlang's behaviour with respect to message reliability, version existence and quiescence. We use Haskell's code reloading runtime facility to build a prototype solution which can upgrade a single process in a module running on a single node without state preservation. We then discuss other approaches which might scale better in a distributed scenario and can be generalized to any Cloud Haskell like system. We conclude with a list of issues to be resolved to develop a general solution to the problem of hot-code reloading in Cloud Haskell.

To my grandfather

Acknowledgements

First, I would like to thank my thesis adviser Dr Amey Karkare. Without him this thesis would not have been possible. He patiently listened to my problems and gave valuable insights. His proposal to consider ideas from proxy based approaches, was very useful.

I'd also like to thank my parents and others in my life who inspire me and support me in all my endeavours.

I would like to thank all the people at `#haskell` on Freenode IRC, who helped me understand various intricacies of the Haskell programming language. The parallel-haskell mailing list in particular was very useful to me during my research.

Special thanks to Tim Watson for helping me with Cloud Haskell on `#haskell-distributed` and sketching out the proxy approach.

Contents

List of Figures viii

1 Introduction 1

1.1 Challenges in a distributed system 2

1.2 Hot Code Reloading 2

1.2.1 The Problem 2

1.2.2 Why Hot Code Reloading 3

1.2.3 Example of Hot Code Reloading in Erlang 4

1.3 Contributions 5

2 A brief tour of Cloud Haskell 7

2.1 The Design Decisions 7

2.1.1 Implementing Erlang in Haskell 7

2.1.2 Library vs Run Time System 8

2.1.3 Modular Architecture 9

2.1.4 The Actor Model and Cloud Haskell 10

2.1.5 Actor vs Thread 10

2.2 The Core API 11

2.3 Ping Pong in Cloud Haskell 12

3 Related work 15

3.1 Formal specification of DSU 15

3.2 Related ideas and techniques 16

3.2.1 Quiescence . . . 16
3.2.2 Binary Code Rewriting . . . 17
3.2.3 Proxies, Intermediaries and Indirection Levels . . . 17
3.2.4 State Transfer and Transformation Functions . . . 18
3.2.5 Source code static analysis . . . 18
3.2.6 Using underlying facilities . . . 18
3.2.7 Version Coexistence . . . 19
3.3 DSU and Functional Programming . . . 20
3.3.1 Haskell . . . 20
3.3.2 Erlang . . . 21

4 An Attempt at Hot Code Reloading 22

4.1 Understanding Erlang's Behaviour . . . 23
4.1.1 Message loss during updates . . . 24
4.1.2 Quiescence and Version Coexistence . . . 24
4.2 An approach using plugins . . . 26
4.2.1 The Plugins Way . . . 26
4.2.2 A minimal plugins example . . . 27
4.2.3 DSU in Ping Pong . . . 29
4.2.4 Unresolved Problems . . . 30
4.3 The Proxy Approach . . . 35
4.3.1 Cloning a Process . . . 36
4.3.2 Relocating a process . . . 36
4.3.3 Addressability and Roaming . . . 36
4.3.4 Other Problems . . . 38
4.3.5 Evaluation . . . 38

5 Conclusions 40

5.1 Implementation Challenges . . . 40
5.2 Future work . . . 41

References **43**

List of Figures

1.1 An example of hot code loading 4

1.2 An example of hot code loading 5

2.1 Cloud Haskell API 12

2.2 Ping Pong in Cloud Haskell 13

4.1 Program to check message loss in Erlang 25

4.2 A minimal plugins example 28

4.3 DSU in Ping Pong - PingPong.hs 31

4.4 DSU in Ping Pong - Main.hs 32

Chapter 1

Introduction

With CPU gigahertz race behind us, parallelism has started rising in importance. When Moore's law used to guarantee doubling CPU clock speed every two years, the lazy strategy to improve the speed of a slow program was to simply wait. With time, it would become automatically faster. But this does not happen anymore. CPU clock speeds have stagnated. Programs have to be re-written to use multiple cores to improve performance. The cost of scaling the number of cores on a multi-core CPU is quite prohibitive. Buying expensive systems with large number of CPU cores is termed as *vertical scaling*. This is neither elastic nor very reliable. *Horizontal scaling* is about using a large number of low cost compute machines and improving performance by simply adding more machines. Horizontally scaling the computation across a range of commodity compute nodes is much more cost effective, scales more incrementally and is becoming increasingly popular for programming compute intensive distributed applications. This approach of renting a cluster of nodes with on-demand elastic scaling of resources is commonly known as "cloud computing". Cloud Haskell [1] is a framework for writing distributed cloud-based applications in Haskell. It is conceptually very similar to Erlang [2] which is very popular in industry [3].

1.1 Challenges in a distributed system

Developing distributed programs which can scale horizontally across a large number of nodes presents some unique challenges:

- When programming the cluster as a whole, there is a need to coordinate various processes running on heterogeneous systems. Most programming languages do not directly address the problem of distributed concurrency. The dominant model of concurrency in most mainstream programming languages is usually the shared-memory variety. It relies on the concept of multiple threads modifying shared mutable data. This model is not very useful in a distributed model. Glasgow Distributed Haskell [4] tries to replicate the model of shared memory concurrency in a distributed system. This model is not very successful because the cost of moving data across in a distributed system becomes a dominant factor. By making message passing explicit, Cloud Haskell exposes the cost of message passing to the programmer.
- The problem of fault tolerance becomes non-trivial in a distributed system. When a distributed program is running across hundreds of thousands of nodes, some of the nodes will fail at any given moment of time with very high probability. A failure of a node should not require restarting the whole calculation , or the calculation might never finish. The programmer needs tools to detect and respond to failures as part of the programming model. Cloud Haskell allows monitoring of processes and exposes primitives to handle node failures.

1.2 Hot Code Reloading

1.2.1 The Problem

Current Cloud Haskell systems cannot be easily upgraded from one version of the code to the next. There is no support in the tool to enable safe upgrades. Moreover,

if there are multiple versions running concurrently, processes having incompatible types won't communicate and the current implementation fails silently. This makes debugging very hard since the programmer cannot figure out the issue if no error is generated due to incompatible versions. Ad-hoc update mechanisms like using external tools for updating a cluster are neither safe nor efficient and very error-prone.

In this thesis, we work on the problem of supporting hot-code reloading, commonly known in literature as Dynamic Software Updating, a running Cloud Haskell system with zero downtime.

1.2.2 Why Hot Code Reloading

- Zero downtime is a fairly common requirement in large scale distributed systems running critical business logic. This is especially true in financial transaction systems, telephone switches, airline traffic control systems and other mission critical systems.

- Hot code reloading is less expensive than using redundant hardware for managing upgrades. Loss of web service during maintenance is no more acceptable and leads to lost revenues. For example, Visa makes use of 21 main frames to run its fifty-million-line transaction processing system. It is selectively able to take machines down and upgrade them by preserving relevant state in other online systems and complex state migration. This approach is expensive as well as increases the complexity of deploying updates.

- It helps in rapid prototyping and increases developer productivity by reducing the length of an iteration cycle. The ability to modify a running program and update it on the fly saves a lot of time which would be otherwise wasted in restarting a program and rebuilding all the relevant state. Moreover, the real time feedback available to the programmer when he changes the program is very valuable in supporting interactive programming.

- Language level updating facility is more reliable than using external tools to

safely update a software. Manually distributing the updated version using tools like scp, puts the burden of responsibility on the programmer to make sure that all nodes are running the same node. This can be quite error-prone because Cloud Haskell would silently fail during communication between processes of incompatible types.

1.2.3 Example of Hot Code Reloading in Erlang

Erlang supports language level dynamic software updating. A process in erlang can update into the new version by making an external call to its module.

We demonstrate an example of hot code reloading in erlang by the help of a counter process. It can receive a message to increment the running counter, or a message to send the current counter value, it can also receive a message to update itself.

Figure 1.1 Hot code loading in erlang : version 1

```
%% A process whose only job is to keep a counter.
%% First version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  receive
     {increment, Count} ->
        loop(Sum+Count);
     {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
     update ->
        ?MODULE:codeswitch(Sum)
        % Force the use of 'codeswitch/1' from the latest MODULE version
  end.

codeswitch(Sum) -> loop(Sum).
```

In version 2, we add the possibility to reset the counter to 0.

Figure 1.2 Hot code loading in erlang : version 2

```
%% Second version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  receive
     {increment, Count} ->
        loop(Sum+Count);
     reset ->
        loop(0);
     {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
     update ->
        ?MODULE:codeswitch(Sum)
  end.

codeswitch(Sum) -> loop(Sum).
```

On receiving an *update* message, loop will execute an external call to codeswitch. If there is a new version of "counter" module in memory, the its codeswitch function will be called with the update state. In our example, we pass the same state to the new version.

The goal of hot code reloading in Cloud Haskell is to have similar style of code upgrade facility as Erlang.

1.3 Contributions

The contributions made in this thesis are:

- Understanding Erlang's behaviour with respect to Version Coexistence, *Quiescence* and message reliability during upgrades.
- A prototype implementation for upgrading a single Cloud Haskell node with multiple processes with one version of module in memory with no state persist-

ence.

- A discussion on other approaches that we tried and the challenges faced.
- A high level overview of the *proxy* approach with emphasis on the problems of addressibility, serialization and code transmission.
- Discovered multiple issues with *plugins* library which need to be fixed to enable hot-code reloading in Cloud Haskell.

The rest of the thesis is organized as follows. Chapter 2 gives a brief tour of Cloud Haskell and the design decisions taken which set the background for understanding the rest of the thesis. Chapter 3 highlights the major related concepts found in the field of Dynamic Software Updating. Chapter 4 describes our various experiments and approaches to hot code reloading in Cloud Haskell. Chapter 5 concludes by listing the important and urgent problems that need to be solved to enable full support for hot-code reloading

Chapter 2

A brief tour of Cloud Haskell

In this chapter, we will briefly cover the overall design decisions which influence the development of Cloud Haskell. This is useful in understanding the trade-offs taken by Cloud Haskell. The problem of hot code reloading and our implementation in the following chapters requires an understanding of these decisions.

2.1 The Design Decisions

2.1.1 Implementing Erlang in Haskell

The key idea is : Program the cluster as a whole, not individual nodes. Same program runs on all the nodes. The programmer does not have to worry about how the individual nodes behave. To implement this model, Cloud Haskell takes inspiration from the Erlang [3] style of programming which has been highly successful in the industry . The design decisions are influenced heavily by the motto: "If in doubt, do it the way Erlang does it".

We give reasons for why Cloud Haskell is an improvement over Erlang:

Improved tooling and library ecosystem Compared to Erlang, Haskell has a much more mature ecosystem of tools and libraries. The package manager of Haskell, Hackage has more than 5000 packages covering a variety of domains.

Static vs Dynamic Typing Haskell's static typing with type inference eliminates whole class of bugs at compile time.

Modular Architecture In erlang, the networking stack is embedded into its run time system. It is difficult to use Erlang in exotic network protocols such as infiniband, CCI, etc Cloud Haskell on the other hand keeps the network stack decoupled as a library.

Multiple Concurrency Abstractions Apart from message passing across nodes, individual nodes can still use concurrency primitives such as Threads and STM as necessary. In case of high capacity, multi-core nodes, it might make better sense to use shared concurrency constructs than actor model and Cloud Haskell allows that. It encourages right abstraction at the right level. Erlang does not provide other concurrency primitives.

A more precise and well-defined semantics Erlang's semantics does not guarantee message reliability. In case of node disconnects, Erlang buffers the messages temporarily and then drops messages, sacrificing reliability property. Since messages cannot be buffered indefinitely, it is difficult to guarantee reliability. Cloud Haskell instead provides an explicit reconnect primitive to accept intermediate message loss.

2.1.2 Library vs Run Time System

Compared to other approaches where the implementation is tied up with the run-time system (RTS), Cloud Haskell is implemented as a Haskell library. The advantages are the following :

Portability As a library, it can be compiled against Haskell compilers other than GHC.

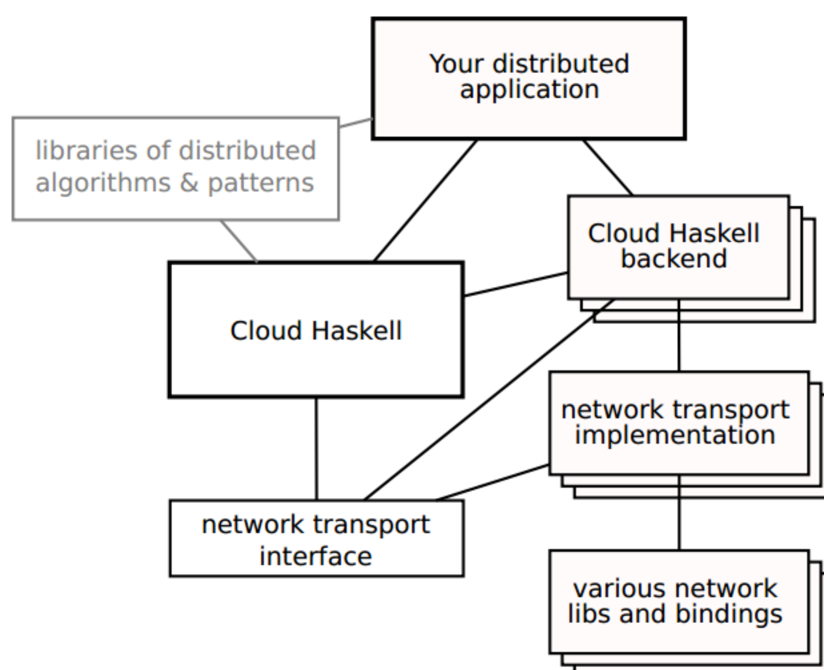
Modular The whole architectural flexibility afforded by Cloud Haskell is only possible because it is not tied with the RTS. Various layers of the stack can be

replaced with alternative implementations. Moreover, alternative implementations of Cloud Haskell can be developed and easily compared with each other.

Contributor Friendly GHC is a huge code-base and monolithic. If cloud haskell was embedded into the run-time system, any contributions from other open-source developers would be difficult due to the higher bar in developing patches for GHC and getting those patches accepted.

Speed of Development GHC releases are usually bi-yearly or longer. Any small improvements in Cloud Haskell would have to wait for six months to get to mainstream if implemented inside the RTS. As a library, the fixes can be uploaded continuously to Hackage without any delay.

2.1.3 Modular Architecture



The architecture of Cloud Haskell is highly decoupled from the transport backend. A major issue with Erlang is that it is difficult to use in exotic network protocols like infiniband, CCI, etc. Cloud Haskell has a unified Network Transport interface which provides a uniform abstraction for a variety of network protocols. Porting a Cloud Haskell program to another network protocol is as simple as using the corresponding

network transport implementation for the given protocol. Even the Cloud Haskell module is a stub API which simply re-exports the actual Cloud Haskell backend API. This makes it very easy to try out competing implementations of Cloud Haskell itself.

2.1.4 The Actor Model and Cloud Haskell

The dominant model of programming a distributed application running in a cluster of nodes is via message-passing. MPI [5] and Actor Model [6] are the popular models for message-passing. In actor model, isolated (no shared memory) lightweight processes are the smallest program primitives [6]. These processes communicate only by sending and receiving messages.

2.1.5 Actor vs Thread

Although, Actor is a concurrency abstraction similar to Thread, it has subtle differences which makes it suitable in a distributed setting. A comparison is listed in the table below.

Table 2.1: Actor vs Thread

Actor	Thread
can create more actors	can create more thread
can have private local state	can have private local state
has NO shared state	has limited shared state
communicates via asynchronous message passing	communicates via shared variables

The essential difference between the two is about isolation and message passing. The appeal of actor model is its simplicity. By avoiding shared state, it eliminates a whole class of concurrency bugs by definition. It is easier to reason about because each actor can now be considered in isolation and independent of other actors.

More formally, an actor is a computational entity that, in response to a message it receives, can concurrently [6] :

- send a finite number of messages to other actors

- create a finite number of new actors
- designate the behavior to be used for the next message it receives

The only popular programming language based on actor model is Erlang [7]. Recently there has been a rise in the number of different implementations of actor model. Instead of designing a new programming language, the current actor based systems are implemented by embedding as a concurrency paradigm inside a host language. This has obvious advantages which are discussed in [8]. Some popular implementations are Akka (Scala and Java), Celluloid (Ruby) and Cloud Haskell (Haskell).

2.2 The Core API

For complete documentation of Cloud Haskell API, the haddock documentation can be read at [9].

The central abstraction in Cloud Haskell for message passing is the Process monad. It keeps track of the state associated with a process, primarily the message queue associated with the process. Any code running in the Process monad has access to the data structures representing the process and can pass these data structures to other Process-monadic functions that it calls. All functions dealing with sending and receiving of messages, spawning and linking processes must be in the Process monad.

- Since Process monad in an instance of MonadIO, arbitrary IO functions can be called in Process monad via liftIO.
- ProcessId is a type corresponding to the process identifier. NodeId is similarly defined for nodes.
- Any type which is an instance of Typeable and Binary is also an instance of Serializable.

- send takes a process-id and a serializable message and sends it asynchronously to the corresponding process. An example in the next section demonstrates the send and receive functions.

Figure 2.1 Core API

```
instance Monad Process
instance MonadIO Process

data ProcessId
data NodeId

class (Typeable a, Binary a) => Serializable a

send :: Serializable a => ProcessId -> a -> Process ()
expect :: Serializable a => Process a

spawn :: NodeId -> Closure (Process ()) -> Process ProcessId

getSelfPid :: Process ProcessId
getSelfNode :: Process NodeId
```

2.3 Ping Pong in Cloud Haskell

The program described in Figure 2.2 is the hello world equivalent in distributed systems. A client process sends a Ping message to the server process. On receiving the Ping, the server process prints a message “Got a Ping”.

In main, we initialize a network transport tcp backend, setup a local node and run the process *ignition* on the local node.

The process “ignition” spawn the “server” process and the “client” process on the local node and waits a while otherwise the child processes will get killed when the parent terminates.

A channel is a tuple (SendPort a, ReceivePort a) on which only values of type “a” can be transmitted. In this case, it is the data type Ping. We create a binary instance of Ping to make it serializable.

Figure 2.2 Ping Pong in Cloud Haskell

```
module Main where
import Control.Concurrent ( threadDelay )
import Data.Binary
import Data.Typeable
import Control.Distributed.Process
import Control.Distributed.Process.Node
import Network.Transport.TCP

data Ping = Ping deriving (Typeable)
-- binary instance declaration of Ping
instance Binary Ping where
    put Ping = putWord8 0
    get      = do { getWord8; return Ping }
-- Ping is now serializable
server rPing = do
    Ping <- receiveChan rPing
    liftIO $ putStrLn "Got a ping!"

client :: SendPort Ping -> Process ()
client sPing =
    sendChan sPing Ping

ignition :: Process ()
ignition = do
    -- start the server
    sPing <- spawnChannelLocal server
    -- start the client
    spawnLocal $ client sPing
    liftIO $ threadDelay 100000 -- wait a while

main :: IO ()
main = do
    Right transport <- createTransport "127.0.0.1" "8080"
                            defaultTCPParameters
    node <- newLocalNode transport initRemoteTable
    runProcess node ignition
```

The client sends Ping on the SendPort of server. In case of server, receiveChan blocks until a message of type Ping is received on the receive port. On receiving the message, server prints “Got a Ping” to the console.

Chapter 3

Related work

The research field concerning hot code reloading in software is known as Dynamic Software Updating (DSU). DSU is a field of research focused on safely upgrading programs while they are running [10]. Although there is no existing research on DSU in the area of distributed functional programming, there is lot of work in the field of imperative and object oriented programming languages. We will briefly describe the main ideas and concepts found in the literature. For a detailed survey of DSU systems, please refer to the survey by Miedes and Muñoz-Escoí[11].

3.1 Formal specification of DSU

Any running program can be thought of a tuple (δ, P) where δ is the current program state and P is the current program code. DSU systems transfer a running program (δ, P) to (δ', P'). The state must be transformed into a representation P' expects. This requires a state transformer function. Thus, DSU transforms (δ, P) to (S(δ), P). An update is considered valid if and only if the running program (S(δ), P') can be reduced to a point tuple (δ', P') that is reachable from the starting point of the new version of the program, (δ_{init}, P'). For a formal description of this *validity*, please refer to the paper by Gupta, Jalote and Barua[12].

Although this is one formal definition for DSU, there is no consensus on the standard definition of DSU which can be applied in all cases. Miedes and Muñoz-

Escoí[11] presents a list of definitions and requirements expected of a DSU system according to different authors.

3.2 Related ideas and techniques

In this section, we briefly discuss the most relevant concepts and techniques related to dynamic software updating. Although most of these issues have been proposed in the context of imperative programming languages, they serve to highlight high level approaches and challenges faced while building any DSU system.

3.2.1 Quiescence

Many proposals for DSU use some form of *quiescence*. The basic idea is that before an update, the component to be updated reaches some *stable* state. Depending on the author, this stability requirement is defined in different ways.

In the paper by Kramer and Magee[13], a node is *quiescent* if it is not going to start data exchange or receive any data from any other node [13]. In [14], the authors argue that when upgrading a subset of nodes, they must be in a quiescent state. Segal and Frieder [15] propose the concept of *active process* and *inactive process*. They propose the criteria for a process to be inactive and argue that when applying an update, the process must be inactive. [12] propose that before updating a function, the execution stack is checked to see if the function is being executed. The update should only be applied if it is not present. Giuffrida and Tanenbaum [16] propose an update manager component. During an update, update manager notifies the components to update. The components transit to a *controlled state* as soon as possible and send a reply back. When the update manager receives all the replies, the update can be applied.

Some authors have criticized the concept of *quiescence* and its blocking requirements. For example, Vandewoude et al. [17] argue that quiescence is stricter than necessary. They propose the concept of *tranquility*[17] as a more relaxed alternative.

3.2.2 Binary Code Rewriting

Some authors propose rewriting of the binary code of the programs in memory, mostly in Java and C. Fabry [18] was one of the first authors to propose the use of binary rewriting in DSU systems. Bytecode rewriting of Java classes was proposed in [19] to build an intermediary level that enable a regular Java application to be updated in runtime. Hicks and Nettles [20] also use binary rewriting techniques to modify the data types, service implementation and the client code that has access to the patched code. Gregersen and Jørgensen [21] used the standard instrumentation facilities provided by the JVM as part of their mechanism of updating Java programs. There are many tools and libraries that offer bytecode manipulation (including runtime manipulation) for Java such as ObjectWeb ASM [22], CGLIB [23], Javassist [24], JRebel [25] and others.

Haskell currently does not have any tooling or infrastructure for binary rewriting or instrumentation of binary code.

3.2.3 Proxies, Intermediaries and Indirection Levels

The idea of redirecting function calls by remapping its handlers is used in [26, 18]. Segal and Frieder [15, 27] use interprocedures, which are intermediate procedures used to redirect the client invocations to old version procedures to their new version counterparts. Purtilo, Hofmeister and Purtilo [28] propose the use of a software bus to connect software modules by means of proxies. These proxies and the bus itself intercept all the function calls and implement the dynamic reconfiguration of the modules. Ajmani, Liskov and Shrira[30] propose the concept of *simulation objects* to represent the past and future versions of an object. These objects are exposed to the client as real objects. But internally they handle all redirection and invocation to the actual objects.

We discuss a similar approach using proxies for Cloud Haskell in section 4.3

3.2.4 State Transfer and Transformation Functions

Several authors identify the need to migrate the state from old version to the new version for consistency. The concept of state transfer was first proposed in [31]. The basic idea is defining two accessor functions *getState* and *setState* to retrieve and set the state of a component. Before upgrading a component, using getState a seriablizable representation of the state is obtained which can then be migrated using some transformation functions supplied by the programmer like in [31, 26, 27, 20, 29] and then transferred to the new version by using setState function. Gregersen and Jørgensen also used a state transformation step which could be applied lazily and on demand [21].

3.2.5 Source code static analysis

A variety of papers related to DSU use some kind of static analysis of the source code to achieve different objectives. For instance, in [32, 33], static analysis is used to identify points in which it is possible to dynamically apply updates to the classes which ensure some correctness property. Neamtiu et al. proposed Ginseng [34], a DSU solution for programs written in C. In there proposal, they use static analysis of the source code to ensure that the updates are type-safe. Moreover, they also use annotated source code to identify safe points which is manually marked by the programmer. Altekar et al. propose OPUS [35] which uses similar analysis to detect unsafe dynamic updates. Authors like Hicks and Nettles [20] and Chen et al. in their POLUS system [36] use static analysis to build a patch that can be applied to the old version dynamically.

3.2.6 Using underlying facilities

A number of papers are based on the underlying features of the given programming language or infrastructure. Bloom's Ph. D. thesis' [26] solution is based on the programs written only in Argus programming language. For C, there are some options

like Gupta, Jalote and Barua [12], Ginseng by Neamtiu et al. [34] and POLUS by Chen et al. [36]. In Java, there is great support for language level DSU facility [37, 21, 38]. Dmitriev [39] uses an existing mechanism available in HotSpot Java Virtual Machine to provide DSU of java code during debugging phase. Gregersen and Jørgensen propose DSU solutions based on modifying the standard JVM.

The other approach is that authors provide their own custom infrastructure or programming language. For example, Kramer and Magee proposal [13, 14] is based on their CONIC configuration language and infrastructure. Stoyle et al. proposal builds its own programming language, compiler, runtime and related tooling in [33].

Some solutions need specific hardware support. The proposal by Frieder and Segal needs that the hardware architecture supports *indirect addressing mode.* Gupta, Jalote and Barua [12] require that the hardware support *segment based memory addressing.*

3.2.7 Version Coexistence

In some proposals, the new version can never co-exist with the old version. This is ensured by asking the program to reach some stable state, performing the update and then uninstalling the old version to prevent it from running at the same time.

The other approach is where multiple versions execute concurrently until all the all clients are updated. For example, Segal and Frieder use *interprocedures* [15] which delegate on the real implementations which can be old as well as new version. Ajmani, Liskov and Shrira propose *simulation objects as proxies* that wrap the real service objects. For a specific service object, multiple *simulation objects* can exist which define the past and future versions of the object. All of them can co-exist and be called by different pieces of client code which may be at different update stages. POLUS [36] also allows old and new versions of the same code as well as data. But, it ensures that *old(new) code can only access old(new) data respectively.*

3.3 DSU and Functional Programming

In the area of functional programming, there is lack of research work compared to the imperative model. More specifically, in distributed functional programming with message-passing semantics like Cloud Haskell, the field of DSU is relatively unexplored.

3.3.1 Haskell

The major and only work in DSU in Haskell programming language is by Stewart and Chakravarty [40]. In his Ph.D. thesis [41], Stewart provided mechanisms for dynamic linking, loading, hot swapping and runtime type checking in Haskell for the first time. The proposed solution is applicable to the problem of dynamic extension in statically typed functional programming languages with type erasure [41]. It develops the concept of *fully dynamic* software architectures, where the static core is minimal and all code is hot swappable. This is different from many other dynamically extensible architectures like Emacs and Linux Kernel which are not *fully dynamic* as some core static functionality cannot be dynamically updated at runtime. They show its feasibility by building two applications : Yi, an extensible editor and Lambdabot, a plugin-based IRC robot, with ability for hot-code reloading as well as dynamic re-configuration and extension via type-safe plugins and embedded DSLs.

The basic primitive is a runtime service provided by Glasgow Haskell Compiler to load compiled Haskell modules into the address space of a running haskell application. A library, called *plugins* [42] for dynamic linking and runtime evaluation of Haskell code, is a major contribution of [41].

We will explore the use of *plugins* library in the next chapter to support hot-code reloading techniques in Cloud Haskell.

3.3.2 Erlang

Armstrong et al. in his book titled, *Concurrent Programming in ERLANG* gives exhaustive details about various features of Erlang including its hot-swapping facility. But it does not talk about the implementation details or any formal understanding of behaviour of dynamic updates in Erlang. The first work in trying to reason about Erlang code was by Armstrong et al. in his Ph.D. thesis [2]. Building on his work, Claessen tried to formalize a semantics for distributed erlang in [43] and improved it further in [44]. But these semantics don't accurately model the behaviour of current implementation of erlang fully. Also there is no discussion in these papers to formalize the dynamic update facility of erlang.

Svensson, Fredlund and Benac Earle[45] proposed a unified semantics for future erlang implementations which behave the same in local and distributed scenarios of erlang and help in understanding a number of poorly understood features of current and future implementations of erlang. It also does not talk anything about DSU semantics in erlang. The current implementation of erlang does not provide any safety guarantees on its code swapping features. Although there are no formal guarantees, a significant number of erlang production deployments using its code-reloading facility and stress its importance as a major benefit of erlang.

Due to lack of any formal semantics for code reloading in erlang, we resort to understanding its behaviour during updates by experimentation or reasoning, which is discussed in the next chapter.

Chapter 4

An Attempt at Hot Code Reloading

While searching for relevant implementations of DSU in distributed functional programming, we found that Erlang's language level hot-swapping [2] is very similar to what we want to achieve. Since Cloud Haskell is already very similar in concept to Erlang and the approach of Cloud Haskell designers has been : "If in doubt, do it the way Erlang does it", our initial approach was to follow on the footsteps of Erlang. We tried to understand exactly how DSU works in Erlang. Even though there is no formal semantics for DSU in Erlang, we found that even the documentation [46] for implementation of Erlang and its specific behaviour during code upgrades was very light on details . Hence, in this thesis, we do not focus on what the formal semantics of Cloud Haskell should be. Imitating Erlang, we prototype, experiment and discuss various approaches and their trade-offs in the context of Cloud Haskell.

The design goal of Cloud Haskell is to be highly decoupled from the runtime system (RTS). Our proposal tries to be independent of the RTS. We do not attempt to make changes to the current RTS. Although we depend on the object loading runtime facility of GHC, we are not keen on approaches which would require changes to the current GHC runtime system.

The overall problem of supporting hot code reloading contains many different

subtle sub-problems. The challenges in solving these problems have been discussed in the Ph.D. thesis by Epstein [47]. In our current approach, we only focus on the problem of upgrading a process running on a single node. This corresponds to the Erlang code-upgrade example shown in subsection 1.2.3. Specifically, we do not consider the problem of coordinating upgrades, rollbacks, sending new code to remote nodes and the problem of communicating processes with incompatible versions. These are auxiliary problems but nevertheless important to the overall DSU experience.

The smallest possible situation is : The ping pong example discussed in section 2.3. How to update the server process from one version to the next? Before we can answer that question, we need to figure out how Erlang upgrades modules.

4.1 Understanding Erlang's Behaviour

According to the Erlang manual [46], it keeps two versions of a module in memory i.e old and current. When a module is upgraded i.e new version of the module has to be brought into memory :

- the old version is discarded from memory
- processes running the old version get killed
- the current version is marked as old
- the new version becomes current

How is an update triggered in Erlang?

- Fully qualified function calls of the form ?MODULE:foo() always refer to the *current* version.
- Non qualified calls such as foo() refer to the version in which they were originally invoked.

- Upgrading a running Erlang process reduces to compiling the module, loading the new version into memory and finally calling the process using its fully-qualified function name.

Before we start to sketch out how upgrades should work in Cloud Haskell, we try to find answers to the following questions:

- What kind of guarantees does Erlang provide with respect to message reliability specifically during an upgrade?
- Why does Erlang keep two versions of every module in memory?

4.1.1 Message loss during updates

The question is : What happens to messages in transit during an update? We figure out the answer by trying a simple experiment using the code shown in Figure 4.1. For a live demonstration of this experiment and related discussion, see [48].

See Figure 4.1 for the program under discussion. When run with N = 10, the client sends an upgrade message to the server along with the *Ping* message on its fifth transmission. The client sends 10 *Pings* and receives 10 *Pongs* confirming that no message was lost during upgrades.

4.1.2 Quiescence and Version Coexistence

In Erlang, can we update a process at arbitrary point in time?

The answer is no. Upgrading requires a fully qualified function call to itself. The process is like a single thread. It cannot do any other computation like listening to a network socket or writing to a file while simultaneously calling its newer version. So, the property of *quiescence* is trivial in Erlang by virtue of its isolated process model. Moreover, since processes can be upgraded at different points in time, there is a possibility of type mismatch when different versions communicate. Erlang leaves it to the programmer who has to make sure that incompatible versions can communicate by writing newer versions with backward-compatibility in mind.

Figure 4.1 Program to check message loss in Erlang during update

```
-module('pingpong').
-compile(export_all).
-import(timer,[sleep/1]).


start(N) ->
    Server = spawn(?MODULE,server,[]),
    _ = spawn(?MODULE,client,[Server,N,0]).

server() ->
    receive
        upgrade ->
            compile:file(?MODULE),
            code:purge(?MODULE),
            sleep(5000),
            code:load_file(?MODULE),
            ?MODULE:server();
        {ping,Cid} ->
            Cid ! pong,
            server()
    end.

client(_,0,C) ->
    io:format("DONE!~n"),
    io:format("Received ~p Pongs!~n",[C]);
client(Server,5,C) ->
    io:format("Sending an upgrade message!~n"),
    Server ! upgrade,
    From = self(),
    Server ! {ping,From},
    io:format("Sent a PING!~n"),
    receive
        pong ->
            io:format("Received a PONG!~n"),
            client(Server,5-1,C+1)
    end;
client(Server,N,C) ->
    sleep(1000),
    From = self(),
    Server ! {ping,From},
    io:format("Sent a PING!~n"),
    receive
        pong ->
            io:format("Received a PONG!~n"),
            client(Server,N-1,C+1)
    end.
```

The smallest unit of compilation in Erlang is a module. If multiple processes are defined and running inside the same module, upgrading the module from one process should not force the other processes to upgrade also. In fact, multiple versions of a module can co-exist in Erlang. But, it is not possible to support processes which refer to arbitrarily old versions of the module. This would require keeping all past versions of a module in memory leading to space leaks with time. Erlang keeps only the current and the previous version of a module in memory. Older processes simply get killed. This is more reliable than just keeping only one version of a module. In single version scenario, other processes from the same module will get killed if one process gets upgraded. This will be terrible in terms of reliability.

4.2 An approach using plugins

Plugins [42] provides an API for compiling and loading new code into a running Haskell application. Our first approach uses plugins to compile and load the next version.

4.2.1 The Plugins Way

When a Haskell program is made dynamic the *plugins* way, it is re-factored so that the program state δ is passed as a parameter to the *main* function of the program. There is a minimal static core with no application logic. It only takes cares of loading the new version of the actual application. All the application logic sits in the *dynamic* part which can be unloaded and reloaded by the static core.

If there is no active reference to a value in Haskell, the GHC garbage collector reclaims it. While reloading the dynamic application, we need to keep a reference to the program state. The only safe place is the static core. Since the whole dynamic application is reloaded, we must return the execution to the static core with a reference to the program state that needs to persisted. We need to define a function called *upgrade* in the static core which is called from the dynamic application when

it is ready to upgrade. The dynamic application passes the state to *upgrade* function. In the *upgrade* function, the old version is unloaded. The next version is compiled and loaded. The upgrade function can apply transformation functions to transform the state for the new version. The *main* symbol of dynamic application is resolved. The (transformed) state is passed to the *new main*. The dynamic application receives the state, short-circuits the initialization steps and continues running the upgraded version.

This is the recommended way to refactor existing applications to make them dynamic.

4.2.2 A minimal plugins example

We show a small example in Figure 4.2 of a Hello World Haskell program called Plugin.hs which is continuously upgraded every 5 seconds after its execution round. You can change the code of Plugin.hs and see the changes after every upgrade in 5 seconds when its new version is executed again.

The Main.hs module of the application is a static core responsible for reloading the actual application. In this case, the application specific code is defined in Plugin.hs. The upgrade function is defined in the static core. It re-compiles Plugin.hs, loads it into memory and then calls the main of Plugin.hs by passing it the value *upgrade*. The main of Plugin.hs runs and calls upgrade at the end which upgrades the Plugin application. We can make changes to the Plugin.hs file and those changes will be reflected in the output after the next upgrade.

In this example, we do not preserve any state of Plugin.hs but it is possible to change the type of upgrade to pass the state of Plugin.hs to the static core. When upgrade executes, it has a reference to the old state , it can transform the old state if required and then can call the new version with (transformed) state.

Figure 4.2 A minimal plugins example

```
-- Plugin.hs
module Plugin (main) where

type DynamicT = IO ()

main :: DynamicT -> IO ()
main upgrade = do
--  print "New version Now!"
  putStrLn "Hello World!"
  putStrLn $ show $ thing
  threadDelay 5000000 -- wait a while
  upgrade

thing :: String
thing = "42"

-- Main.hs
module Main (main) where

import System.Plugins
import Control.Monad

upgrade :: IO ()
upgrade = do
  r <- makeAll "Plugin.hs" []
  case r of
    MakeFailure msgs -> putStrLn "failed to make" >> print msgs
    MakeSuccess mc fp -> do
      mv <- load fp [] [] "main"
      case mv of
        LoadFailure msgs -> putStrLn "fail" >> print msgs
        LoadSuccess m v -> print "Upgrade Done!" >> v upgrade

type DynamicT = IO ()

main :: IO ()
main = upgrade
```

4.2.3 DSU in Ping Pong

We take the experiment performed in Erlang in Figure 4.1 and port it to Cloud Haskell. Similar to the minimal plugins example, we have a static core here in Main.hs with an upgrade function to upgrade our server process. We cannot follow the *plugins* way in Cloud Haskell. We cannot extract the state of all processes and preserve that state in the static core. Moreover, we must then short-circuit all the transport initialization and process creation steps and resume from where we left of. To be able to do this, we need a mechanism to extract the message queues and set the message queues for individual processes. This is currently not possible in Cloud Haskell. Therefore, we try a different approach. We directly pass the top level symbol that we want to return to after upgrading , i.e *server* in this case. See Figure 4.3 and Figure 4.4 for the source code of our approach.

When we run this program, we find that on the fifth *Ping* from the client, the server *upgrades* and shows the changes made but then the whole application crashes silently probably due to a segmentation fault. There is no debug info. We believe the problem is probably related to keeping only one version of the module in memory. To investigate this, we look into the source of *plugins* package. We find that it indeed does not support loading multiple versions of the same module. Hence, the client process cannot continue since it does not have the old object code in memory. Its *program counter* is pointing to a junk value. It is like *sweeping the rug from underneath*.

If we can change the loading mechanism in *plugins* to support *version coexistence*, it would resolve the above problem. This is essential if we want to have reliable code reloading in Cloud Haskell.

The other problem with this approach is that the message queues and internal process state is not being shared with the *new server*. This is because the *new server* is not called from inside the *old server*. It is called from the upgrade function which does not have any internal state of the *old server*. This is like running a new server from scratch without state persistence.

4.2.4 Unresolved Problems

In the approach described in subsection 4.2.3, we evaluate the call to *new* server at end of upgrade. The type of upgrade is

```
type DynamicT = IO ()
upgrade :: IO ()
```

The type of server is

```
server :: DynamicT -> Process ()
server :: IO () -> Process ()
server upgrade :: Process ()
```

This should result in a compile time type error. But in practice, it compiles successfully. This is because of two reasons. The return type of *load* is polymorphic. The GHC dynamic loader is unsafe. [41]. The compiler does not know the type of the symbol *server* since it does not have the code of *server* at compile time. Since, the return type of load is polymorphic, and the type of the *upgrade* is IO (), the types now become :

```
type DynamicT = IO ()
upgrade :: IO ()
-- v refers to the value of symbol ``server''
v upgrade :: a
-- the type is instantiated to IO ()
v upgrade :: IO ()
v :: IO () -> IO ()
server :: IO () -> IO ()
```

Type checking succeeds, but it will most likely crash at run-time since the type of *server* is obviously wrong. The strange behaviour is that if we execute the upgrade with no change in the source code, the server continues to receive *Pings* and process

Figure 4.3 DSU in Ping Pong - PingPong.hs

```
-- PingPong.hs
module PingPong where ...
...

server :: DynamicT -> Process ()
server st = do
    (cid,x) :: (ProcessId,Int) <- expect
    send cid x
    case x of
      5 -> do
        liftIO $ st
      _ -> do
        server st

client :: DynamicT -> Int -> ProcessId -> Process ()
client st 10 sid = return ()
client st c sid = do
  me <- getSelfPid
  send sid (me,c)
  (v :: Int) <- expect
  client st (c+1) sid

ignition :: DynamicT -> Process ()
ignition st= do
    sid <- spawnLocal $ server st
    cid <- spawnLocal $ client st 0 sid
    liftIO $ threadDelay 100000 -- wait a while

type DynamicT = IO ()
main :: DynamicT -> IO ()
main st = do
    Right transport <- createTransport "127.0.0.1" "8080"
                            defaultTCPParameters
    node <- newLocalNode transport initRemoteTable
    runProcess node (ignition st)
    closeTransport transport
```

Figure 4.4 DSU in Ping Pong - Main.hs

```
-- Main.hs
upgrade :: IO ()
upgrade = do
  r <- makeAll "PingPong.hs" []
  case r of
    MakeFailure msgs -> putStrLn "failed to make" >> print msgs
    MakeSuccess mc fp -> do
      mv <- load fp [] [] "server"
      case mv of
        LoadFailure msgs -> putStrLn "fail" >> print msgs
        LoadSuccess m v -> do
        unloadAll m
        v upgrade


type DynamicT = IO ()

main :: IO ()
main = upgrade
```

them after it is upgraded. This requires further investigation and the Cloud Haskell maintainers have been notified about this issue.

Due to segfaults in the previous approach, we try a different approach where the *new server* is called from the old server. How do we do this? Instead of calling the new server inside upgrade, we can return it as a value and let the *old server* call the *new server*. This is better than previous approach. Calling a Process () monadic action inside another Process () action leads to sharing of the message queues and internal process state. The *bind* operator of the Process monad hides all the *magic*. This is how a recursive loop in server works. When the server calls itself at the end, it *shares* the internal state with the new call.

Lets try to see if we can return *new server* to the *old server* after upgrading the code. Then we can call the *new server* from the *old*. This will be exactly similar to calling *new server* in Erlang. There, we just used a fully qualified function call and the Erlang VM redirected it to the new version automatically.

For this problem, we reason using types. *upgrade* now returns the value of *new server*. But it might not have any value to return in case there is no reloading (no change in the source code. So, we wrap it inside a *Maybe*. The new type of upgrade becomes

```
type DynamicT = IO (Maybe a)
upgrade :: DynamicT
upgrade :: IO (Maybe a)
```

upgrade returns Nothing in case of no change in code or Just server otherwise. Here, a is instantiated to the *type* of server. The type of server is

```
server :: DynamicT -> Process ()
```

This leads to the following type for DynamicT:

```
type DynamicT = IO (Maybe (DynamicT -> Process ()) )
```

On compiling, we get a "Cycle in type synonym declarations" error which is expected. In this approach, this problem is yet to be resolved. Since, we have to return the value of the process and the process also need to take *upgrade* as a parameter, there is a cyclic dependency which cannot be easily resolved.

We reason that if we want to return the symbol for *new server* inside the *old server*, we cannot get around "Cycle in type synonym declarations" error:

- We cannot import the *plugins* library in the dynamic part i.e PingPong.hs. When PingPong.hs is reloaded, all the modules imported by PingPong.hs are unloaded and reloaded in turn. Since upgrade function needs functions from the *plugins* library, we cannot define the *upgrade* function in the dynamic application. The only place we can define it is in Main.hs.

- Since we need upgrade function inside the server process, can we import Main.hs into PingPong.hs and use *upgrade* function without passing it as a parameter to server? This is not possible because we cannot import Main.hs

into PingPong.hs. When PingPong.hs is unloaded, it will unload Main which will unload *plugins*. If *plugins* module is unloaded, we cannot reload anything.

- The only remaining way is to pass the *upgrade* function as a value to PingPong.hs functions. But if *upgrade* function needs to return the value of the *new server*, we would get a "Cycle in type synonym declarations" error.

- If we do not insist that *upgrade* function return the value of the *new server*. The above error will be resolved. In fact, if the upgrade function *never returns*, but simply evaluates the value of the *new server*, it will be correct. This is the approach taken in the *plugins* way. But, in this approach, we cannot share the internal process state with the *new server*. We need to capture all the process state and pass it to upgrade.

- To follow the plugins way and solve the problem of "Cycle in type synonym declarations", Cloud Haskell needs to be internally restructured, so that it is possible to get and set internal process state.

In the current approach, what happens to the message queue of the process that is being upgraded? If we do not make changes to the source, the old version and the new version are same. The call to upgrade reloads the same code and calls the *new server*. This works as expected and continues to send and receive messages. The messages received during the upgrade remain in the message queue and are acted upon by the new server. This implies that there is no message loss during the upgrade. If we make changes to the source, the application crashes due to reasons discussed above. But we believe that the message queue is still receiving the messages during the upgrade and no messages are lost. The reason for this is the following. Every node has a special process called the *node controller*. Among other things, it is responsible for receiving the messages on behalf of all process running on the node and then forward it to the mailbox of the intended process. As long as the process is not killed, its mailbox will not be garbage collected. Since messages are received

by the *node controller* running on another thread, during an upgrade, the incoming messages will not be discarded.

There are a couple of issues that we identified while using *plugins* library for this project:

- The *plugins* library uses *.hi files* for module information and *dependency chasing*. It parses these *.hi files* before loading.new version. This parsing is broken in 64 bit architecture. Our example in 4.3 has been tested to work only on X86 architecture.
- *unload* function, which is used to unload modules from the address space, does not work in GHC 7.6. Although this has been fixed for static builds by Simon Marlow in the next version of GHC, static builds would be longer to build and take more space in general.
- Since *plugins* library uses GHC compiler to do run-time compilation, it currently links the GHC compiler code to the static core. This increases the size of the application binary from less than 1 MB to anywhere from 37 to 66 MB. There is no way current to prevent the compiler from being linked.

Overall the GHC runtime infrastructure for code-reloading and *plugins* in particular are not widely used in production and have subtle bugs which are hard to debug because of lack of good error messages and logging. Based on our experience, we feel that there is lot of scope for improvement in this area.

4.3 The Proxy Approach

In this section, we will briefly discuss about a different approach based on the ideas of indirection or proxies based on subsection 3.2.3.

In our previous approaches, we did not extract the state of a process. Its state was implicit in the Process Monad (which is itself implemented as a State Monad).

The internal state of the Process is *serializable*. Hence, it should be possible to define accessor functions like getState and setState to receive and set the state of a process.

4.3.1 Cloning a Process

Based on the primitives defined earlier, a function can be defined which will create a *clone* of the old process with all the state intact and continue executing from the new version of the module. But the old process instance must be killed before the new process executes. From the perspective of the node controller, it should conceptually look as if the process never died.

The type for clone will be:

```
oldProcess :: a -> Process ()
clone :: ProcessState -> a -> Process ()
```

Here ProcessState is the internal state of the old process. "a" refers to the type (polymorphic) of the parameters passed to the old process.

4.3.2 Relocating a process

We can build a primitive for migrating a process between nodes. The type for relocate will be:

```
relocate :: (Serializable a) => a -> NodeId -> Process ()
```

Here, "a" is instantiated to the type of internal process state (which was extracted using getState). It takes the NodeId to which it should relocate. This approach has the advantage that it allows us to transparently relocate processes based on the resource usage and load on different nodes.

4.3.3 Addressability and Roaming

To send a message to a process, only its ProcessId is required. Its NodeId is part of its ProcessId type. When a process sends a message to another process, the message

is sent based on the NodeId of the ProcessId. The node-controller forwards the message to the intended process. When a process relocates, it migrates from its *home node* to a *remote node.* This is similar to *roaming of users* in cellular networks. In cellular networks, roaming users are monitored by building tables of addresses of currently roaming users.

Before a process relocates, it must notify its *home node-controller* about its new NodeId. Moreover, every node-controller should have a roaming registry, which keeps track of these movements. When other processes, send a message to a relocated process, the messages are initially sent to the home node-controller since the ProcessId they have refers to the old NodeId. The home node-controller needs to forward this message to the current node-controller where the process resides. This leads to one level of indirection. But if the nodes are far apart, this increases the latency and bandwidth requirements. This indirection cannot be reduced. Since there might be old values of ProcessId lingering in the system which would refer to the home NodeId.

One problem is that this approach has a race condition. What would happen if a process which has initially relocated once, relocates again. After the relocation, it notifies its home node-controller. But before the notification reaches home node-controller, say a message expected for this process reaches its home node-controller. Since the home node-controller's entry has not been updated, it will forward the message to the node from where it just relocated. This message will be lost since the process does not exist on that node. One simple solution is that before relocating, the process should also notify its local node-controller about the change. This node controller can then keep the (from -> to) mapping of its relocation until a timer expires. An appropriate worst case expiry time can be just a few seconds. The only disadvantage is that now there can be a chain of hops that a message has to go through if the process is hopping through a series of different nodes very rapidly. But this scenario is purely hypothetical and unlikely to happen in a real problem.

4.3.4 Other Problems

Another major problem is decoding. We cannot used the old functions of *Binary* typeclass in the target node to decode the incoming state. The Binary instance to decode the new state does not yet exist on the target node if it has not been upgraded. Hence, decoding will fail on the target node. This problem has been well documented in [47]. One possible solution is to make it the user's problem. The user should provide transformer functions which take the *ByteString* of the old version and convert it to the new version. Even in Erlang, *gen server* provides callbacks so that the developer can provide these state transformation functions himself. These transformation functions must exist even in nodes which are yet to be updated. One way to achieve this is to break the update into two steps. First update only contains these transformation functions and no type changes. In the next update, type changes can be sent which can be decoded by using the transformation functions sent in the first update.

The other problem is about sending the code of new version to remote nodes. Although code can be sent to other nodes as a new type of message, the main issue is to compile and bring it to the address space of the application. Here, the only solution is to build every instance with *plugins* support so that new code can be evaluated at run-time using *plugins eval* function. There is no easy way to do it without depending on *plugins* library. Therefore, it is necessary that the *plugins* library be robust and support version co-existence.

4.3.5 Evaluation

Our initial approach does not work well even in a single node scenario. There is no state persistence as the upgraded process does not share the state of the old process. But in the *proxy* approach, we propose primitives to extract the internal state of a process which allows us to create clones of a process. This also enables us to get around problem of "cyclic type synonyms". We do not need to pass the value of *new*

server to the *old server*. We can instead clone the process using the internal state of the server. This approach is also suitable in writing applications in the *plugins* way where all internal state of processes need to be extracted and preserved via static core during upgrades.

One disadvantage is the indirection of messages to home nodes when the processes have moved to a new node. Although, we propose solutions to this problem, it increases the complexity of the node-controller. Since the node-controller is a very important part of managing the node, any additional complexity which is not very essential should not be added to the node-controller. The performance of the node-controller is also the bottleneck in many cases. Increasing its complexity might lead to poor performance. In the *proxy* approach, we still rely on *plugins* for dynamic linking and code loading. There are issues with *plugins* which have been discussed in subsection 4.2.4.

Building and maintaining internal tables of roaming processes might require lot of changes to the source code of node-controller. We have not focused on how to implement these changes in Cloud Haskell. This is left as a future work. We hope that the high-level ideas discussed in this approach guide in implementing the *proxy* approach. An implementation based on this approach needs be built to at least demonstrate the feasibility of this approach.

Chapter 5

Conclusions

First, we laid the ground work by describing the problem and the motivations for achieving hot-code reloading in Cloud Haskell. Then we gave a brief overview of Cloud Haskell with a small example to understand Cloud Haskell applications. After that we did an exhaustive classification of all the related work in the field of Dynamic Software Updating in imperative languages based on the related concepts and techniques used. We also briefly covered the state of the art in DSU in Erlang and Haskell.

We then proposed our approach which relies on *plugins* to reload a process running on a single node. We demonstrate an example of Ping Pong game between clients and servers. Then we discussed the problems with our current approach in subsection 4.2.4. We also proposed another approach based on the concept of Proxies which does not have "cyclic type synonyms" issue. We discuss its advantages and disadvantages in subsection 4.3.5.

5.1 Implementation Challenges

The major implementation challenges we faced are:

- Lack of documentation on how Erlang's code reloading is implemented. A documentation of how Erlang implements code reloading can be a good case study of implementation challenges in DSU in functional distributed programming.

- The lack of good debugging messages in *plugins* library. Lot of time was wasted in figuring out that *.hi parsing* was broken in 64 bit architecture.
- Any restructuring of Cloud Haskell code requires complete understanding of how various modules in Cloud Haskell interact. CLOC[49] says that Cloud Haskell code is organized in 36 Haskell files and 5991 source lines of code. We found documentation of the modules to be lacking. Since there is no high level documentation of how Cloud Haskell code is structured, reading source code is the only way to understand it. Moreover, its not easy to isolate state which is distributed across different modules in Cloud Haskell without coupling them together and rearranging all code across all modules.
- *plugins* library is neither actively maintained nor used by any major Haskell projects. It is used by 13 packages on Hackage but most of them are small applications, last updated three to six years ago and are not actively used in production. We believe the *plugins* way of restructuring all code to pass the program state around is not the best way to enable DSU in existing applications. The reason no Haskell project uses DSU is because it is neither very low-friction nor very easy to integrate. It should not required changing the architecture of code. Some form of annotation to mark the state that needs to be preserved would be ideal. The ecosystem for DSU is nascent in Haskell compared to Java and C. Additional ways to achieve DSU that are easier to use in existing applications and does not require drastic changes to the architecture of existing applications, should increase its value and usage.

5.2 Future work

There are many problems which need to be solved before the goal of building a full fledged hot-swappable version of Cloud Haskell can be realized. Some of them include :

- Adding support for loading multiple versions of the same module in memory

using *plugins*. Both of the proposed approaches need multiple version support from *plugins* library. The *plugins* project has not been maintained since quite a few years. Only after multiple version loading is available in *plugins*, we can hope to provide code reloading in Cloud Haskell as reliably as Erlang. Other non-critical issues mentioned in subsection 4.2.4 should be fixed to make the library more robust.

- Building an implementation based on the ideas of Proxy approach in the context of Cloud Haskell. This will not only solve the "cyclic type synonyms" problem, it will be more flexible and provide a better way to manage resources of different nodes.

- Effective benchmarks for the static version and dynamic version of Cloud Haskell to understand the performance and space penalties of using the dynamic version. This is very essential if we want to use dynamic Cloud Haskell in production.

References

[1] Jeff Epstein, Andrew P. Black and Simon Peyton-Jones. “Towards Haskell in the Cloud”. In: *Proceedings of the 4th ACM Symposium on Haskell*. Haskell ’11. New York, NY, USA: ACM, 2011, pp. 118–129. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034690. URL: http://doi.acm.org/10.1145/2034675.2034690 (visited on 27/05/2014).

[2] Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams. *Concurrent Programming in ERLANG*. 1993.

[3] *Erlang (programming language)*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 608888384. 24th May 2014. URL: https://en.wikipedia.org/w/index.php?title=Erlang_(programming_language)&oldid=608888384 (visited on 01/06/2014).

[4] Robert F. Pointon, Philip W. Trinder and H.-W. Loidl. “The design and implementation of Glasgow Distributed Haskell”. In: *Implementation of Functional Languages*. Springer, 2001, pp. 53–70. URL: http://link.springer.com/chapter/10.1007/3-540-45361-X_4 (visited on 01/06/2014).

[5] William Gropp, Ewing Lusk and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999. URL: http://books.google.co.in/books?hl=en&lr=&id=xpBZ0RyRb-oC&oi=fnd&pg=PA1&dq=mpi&ots=ub9ro0OJ9_&sig=7mwhI8BTmLqt_nvsZRzywqO-YyA (visited on 01/06/2014).

[6] Carl Hewitt, Peter Bishop and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804 (visited on 01/06/2014).

[7] Gul Agha. *An overview of actor languages*. Vol. 21. 10. ACM, 1986. URL: http://dl.acm.org/citation.cfm?id=323743 (visited on 01/06/2014).

[8] Rajesh K. Karmani, Amin Shali and Gul Agha. “Actor frameworks for the JVM platform: a comparative analysis”. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20. URL: http://dl.acm.org/citation.cfm?id=1596658 (visited on 01/06/2014).

[9] *Control.Distributed.Process*. URL: http://hackage.haskell.org/package/distributed-process-0.4.2/docs/Control-Distributed-Process.html (visited on 27/05/2014).

[10] Gavin Bierman, Michael Hicks, Peter Sewell and Gareth Stoyle. “Formalizing Dynamic Software Updating”. In: 2003, pp. 13–23.

[11] Emili Miedes and F. D. Muñoz-Escoí. *A survey about dynamic software updating.* Tech. Rep. ITI-SIDI-2012/003, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, 2012. URL: http://web.iti.upv.es/~fmunyoz/research/pdf/TR-ITI-SIDI-2012003.pdf (visited on 30/05/2014).

[12] D. Gupta, P. Jalote and G. Barua. “A formal framework for on-line software version change”. In: *IEEE Transactions on Software Engineering* 22.2 (Feb. 1996), pp. 120–131. ISSN: 0098-5589. DOI: 10.1109/32.485222.

[13] Jeff Kramer and Jeff Magee. “Dynamic Configuration for Distributed Systems”. In: *IEEE Trans. Softw. Eng.* 11.4 (Apr. 1985), pp. 424–436. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232231. URL: http://dx.doi.org/10.1109/TSE.1985.232231 (visited on 01/06/2014).

[14] Jeff Kramer and Jeff Magee. “The Evolving Philosophers Problem: Dynamic Change Management”. In: *IEEE Trans. Softw. Eng.* 16.11 (Nov. 1990), pp. 1293–1306. ISSN: 0098-5589. DOI: 10.1109/32.60317. URL: http://dx.doi.org/10.1109/32.60317 (visited on 01/06/2014).

[15] M.E. Segal and O. Frieder. “Dynamically updating distributed software: supporting change in uncertain and mistrustful environments”. In: *, Conference on Software Maintenance, 1989., Proceedings.* , Conference on Software Maintenance, 1989., Proceedings. Oct. 1989, pp. 254–261. DOI: 10.1109/ICSM.1989.65219.

[16] Cristiano Giuffrida and Andrew S. Tanenbaum. *A Taxonomy of Live Updates.*

[17] Yves Vandewoude, Peter Ebraert, Yolande Berbers and Theo D’Hondt. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Trans. Softw. Eng.* 33.12 (Dec. 2007), pp. 856–868. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70733. URL: http://dx.doi.org/10.1109/TSE.2007.70733 (visited on 01/06/2014).

[18] R. S. Fabry. “How to Design a System in Which Modules Can Be Changed on the Fly”. In: *Proceedings of the 2Nd International Conference on Software Engineering.* ICSE ’76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 470–476. URL: http://dl.acm.org/citation.cfm?id=800253.807720 (visited on 30/05/2014).

[19] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana and Giuseppe Ursino. “Handling Run-time Updates in Distributed Applications”. In: *Proceedings of the 2005 ACM Symposium on Applied Computing.* SAC ’05. New York, NY, USA: ACM, 2005, pp. 1375–1380. ISBN: 1-58113-964-0. DOI: 10.1145/1066677.1066987. URL: http://doi.acm.org/10.1145/1066677.1066987 (visited on 30/05/2014).

[20] Michael Hicks and Scott Nettles. “Dynamic Software Updating”. In: *ACM Trans. Program. Lang. Syst.* 27.6 (Nov. 2005), pp. 1049–1096. ISSN: 0164-0925. DOI: 10.1145/1108970.1108971. URL: http://doi.acm.org/10.1145/1108970.1108971 (visited on 30/05/2014).

[21] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. "Dynamic update of Java applications—balancing change flexibility vs programming transparency". In: *Journal of Software Maintenance and Evolution: Research and Practice* 21.2 (1st Mar. 2009), pp. 81–112. ISSN: 1532-0618. DOI: 10.1002/smr.406. URL: http://onlinelibrary.wiley.com/doi/10.1002/smr.406/abstract (visited on 30/05/2014).

[22] Eric Bruneton, Romain Lenglet and Thierry Coupaye. "ASM: A code manipulation tool to implement adaptable systems". In: *In Adaptable and extensible component systems*. 2002.

[23] *cglib/cglib*. GitHub. URL: https://github.com/cglib/cglib (visited on 30/05/2014).

[24] *Javassist*. URL: http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/ (visited on 30/05/2014).

[25] *JRebel*. zeroturnaround.com. URL: http://zeroturnaround.com/software/jrebel/ (visited on 30/05/2014).

[26] Toby Bloom. "Dynamic Module Replacement in a Distributed Programming System". In: *in a Distributed Programming System, MIT-LCSTR -303*. 1983.

[27] Ophir Frieder and Mark E. Segal. "On Dynamically Updating a Computer Program: From Concept to Prototype". In: *J. Syst. Softw.* 14.2 (Feb. 1991), pp. 111–128. ISSN: 0164-1212. DOI: 10.1016/0164-1212(91)90096-O. URL: http://dx.doi.org/10.1016/0164-1212(91)90096-O (visited on 30/05/2014).

[28] James M. Purtilo. "The POLYLITH Software Bus". In: *ACM Trans. Program. Lang. Syst.* 16.1 (1994), pp. 151–174. ISSN: 0164-0925. DOI: 10.1145/174625.174629. URL: http://doi.acm.org/10.1145/174625.174629 (visited on 30/05/2014).

[29] Christine R. Hofmeister and James M. Purtilo. "A Framework for Dynamic Reconfiguration of Distributed Programs". In: *In Proceedings of the 11th International Conference on Distributed Computing Systems*. 1993, pp. 560–571.

[30] Sameer Ajmani, Barbara Liskov and Liuba Shrira. "Modular Software Upgrades for Distributed Systems". In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 452–476. ISBN: 3-540-35726-2, 978-3-540-35726-1. DOI: 10.1007/11785477_26. URL: http://dx.doi.org/10.1007/11785477_26 (visited on 30/05/2014).

[31] Maurice P. Herlihy and Barbara Liskov. "A Value Transmission Method for Abstract Data Types". In: *ACM Trans. Program. Lang. Syst.* 4.4 (Oct. 1982), pp. 527–551. ISSN: 0164-0925. DOI: 10.1145/69622.357182. URL: http://doi.acm.org/10.1145/69622.357182 (visited on 31/05/2014).

[32] Yogesh Murarka and Umesh Bellur. "Correctness of Request Executions in Online Updates of Concurrent Object Oriented Programs". In: *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. APSEC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 93–100. ISBN: 978-0-7695-3446-6. DOI: 10.1109/APSEC.2008.33. URL: http://dx.doi.org/10.1109/APSEC.2008.33 (visited on 31/05/2014).

[33] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell and Iulian Neamtiu. “Mutatis Mutandis: Safe and Predictable Dynamic Software Updating”. In: *ACM Trans. Program. Lang. Syst.* 29.4 (Aug. 2007). ISSN: 0164-0925. DOI: 10.1145/1255450.1255455. URL: http://doi.acm.org/10.1145/1255450.1255455 (visited on 31/05/2014).

[34] Iulian Neamtiu, Michael Hicks, Gareth Stoyle and Manuel Oriol. “Practical Dynamic Software Updating for C”. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. New York, NY, USA: ACM, 2006, pp. 72–83. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991. URL: http://doi.acm.org/10.1145/1133981.1133991 (visited on 31/05/2014).

[35] Gautam Altekar, Ilya Bagrak, Paul Burstein and Andrew Schultz. “OPUS: Online Patches and Updates for Security”. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 19–19. URL: http://dl.acm.org/citation.cfm?id=1251398.1251417 (visited on 31/05/2014).

[36] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang and Pen-Chung Yew. “POLUS: A POwerful Live Updating System”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 271–281. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.65. URL: http://dx.doi.org/10.1109/ICSE.2007.65 (visited on 30/05/2014).

[37] Filippo Banno, Daniele Marletta, Giuseppe Pappalardo and Emiliano Tramontana. “Handling Consistent Dynamic Updates on Distributed Systems”. In: *Proceedings of the The IEEE Symposium on Computers and Communications*. ISCC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 471–476. ISBN: 978-1-4244-7754-8. DOI: 10.1109/ISCC.2010.5546542. URL: http://dx.doi.org/10.1109/ISCC.2010.5546542 (visited on 30/05/2014).

[38] Tobias Ritzau and Jesper Andersson. “Dynamic Deployment of Java Applications”. In: *IN JAVA FOR EMBEDDED SYSTEMS WORKSHOP*. 2000.

[39] M. Dmitriev. “Towards flexible and safe technology for runtime evolution of java language applications”. In: *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*. 2001.

[40] Don Stewart and Manuel M. T. Chakravarty. “Dynamic Applications from the Ground Up”. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell ’05. New York, NY, USA: ACM, 2005, pp. 27–38. ISBN: 1-59593-071-X. DOI: 10.1145/1088348.1088352. URL: http://doi.acm.org/10.1145/1088348.1088352 (visited on 01/06/2014).

[41] Don Stewart. “Dynamic extension of typed functional languages”. PhD thesis. PhD thesis, University of New South Wales, 2010. URL: http://unsworks.unsw.edu.au/fapi/datastream/unsworks:9098/SOURCE02 (visited on 01/06/2014).

[42] *Hackage: plugins: Dynamic linking for Haskell and C objects*. URL: http://hackage.haskell.org/package/plugins (visited on 01/06/2014).

[43] Koen Claessen. “A semantics for distributed erlang”. In: *In Proceedings of the ACM SIPGLAN 2005 Erlang Workshop*. ACM Press, 2005, pp. 78–87.

[44] Hans Svensson. “A more accurate semantics for distributed Erlang”. In: *In Proceedings of the ACM SIPGLAN 2007 Erlang Workshop*. 2007.

[45] Hans Svensson, Lars AAke Fredlund and Clara Benac Earle. “A Unified Semantics for Future Erlang”. In: *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*. Erlang ’10. New York, NY, USA: ACM, 2010, pp. 23–32. ISBN: 978-1-4503-0253-1. DOI: 10.1145/1863509.1863514. URL: http://doi.acm.org/10.1145/1863509.1863514 (visited on 27/05/2014).

[46] *Erlang – Compilation and Code Loading*. URL: http://www.erlang.org/doc/reference_manual/code_loading.html (visited on 02/06/2014).

[47] Jeffrey Epstein. “Functional programming for the data centre”. PhD thesis. MS thesis, University of Cambridge, 2011. URL: http://msr-waypoint.com/en-us/um/people/simonpj/papers/parallel/epstein-thesis.pdf (visited on 27/05/2014).

[48] Pankaj More. *Do messages get lost when erlang modules are upgraded?* Pankaj More. URL: http://pankajmore.in/code-reloading-in-erlang.html (visited on 03/06/2014).

[49] *CLOC – Count Lines of Code*. URL: http://cloc.sourceforge.net/ (visited on 06/06/2014).

[50] *Well-Typed - The Haskell Consultants: A Cloud Haskell Appetiser (Parallel Haskell Digest 11)*. URL: http://www.well-typed.com/blog/68/ (visited on 27/05/2014).

[51] *Dynamic Software Updating*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 608555677. 21st May 2014. URL: https://en.wikipedia.org/w/index.php?title=Dynamic_Software_Updating&oldid=608555677 (visited on 30/05/2014).

[52] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang and Pen-Chung Yew. “Dynamic Software Updating Using a Relaxed Consistency Model”. In: *IEEE Trans. Softw. Eng.* 37.5 (Sept. 2011), pp. 679–694. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.79. URL: http://dx.doi.org/10.1109/TSE.2010.79 (visited on 30/05/2014).

[53] Eugene Kuleshov. *Using the ASM framework to implement common Java bytecode transformation patterns*. 2007.

[54] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba and Kozo Itano. “A Bytecode Translator for Distributed Execution of ”Legacy” Java Software”. In: Springer-Verlag, 2001, pp. 236–255.

[55] *Apache Commons BCEL™* -. URL: https://commons.apache.org/proper/commons-bcel/ (visited on 30/05/2014).

[56] Allan Raundahl Gregersen, Douglas Simon and Bo Nørregaard Jørgensen. “Towards a Dynamic-update-enabled JVM”. In: *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*. RAM-SE ’09. New York, NY, USA: ACM, 2009, 2:1–2:7. ISBN: 978-1-60558-548-2. DOI: 10.1145/1562860.1562862. URL: http://doi.acm.org/10.1145/1562860.1562862 (visited on 30/05/2014).

[57] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger and David Walker. “Abstractions for Network Update”. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.* SIGCOMM ’12. New York, NY, USA: ACM, 2012, pp. 323–334. ISBN: 978-1-4503-1419-0. DOI: 10.1145/2342356.2342427. URL: http://doi.acm.org/10.1145/2342356.2342427 (visited on 30/05/2014).

[58] Marcin Solarski, Aus Kraków, Vorsitzender Prof and Dr-ing Stefan Jähnichen. *Dynamic Upgrade of Distributed Software Components Promotionsausschuss: vorgelegt von Dipl.-Ing.*

[59] Marcin Solarski and Hein Meling. “Towards Upgrading Actively Replicated Servers On-the-Fly”. In: *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment.* COMPSAC ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 1038–1046. ISBN: 0-7695-1727-7. URL: http://dl.acm.org/citation.cfm?id=645984.675551 (visited on 31/05/2014).

[60] Nigamanth Sridhar, Scott M. Pike and Bruce W. Weide. “Dynamic Module Replacement in Distributed Protocols”. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems.* ICDCS ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 620–. ISBN: 0-7695-1920-2. URL: http://dl.acm.org/citation.cfm?id=850929.851923 (visited on 31/05/2014).

[61] André Pang, Don Stewart, Sean Seefried and Manuel M. T. Chakravarty. “Plugging Haskell in”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell.* Haskell ’04. New York, NY, USA: ACM, 2004, pp. 10–21. ISBN: 1-58113-850-4. DOI: 10.1145/1017472.1017478. URL: http://doi.acm.org/10.1145/1017472.1017478 (visited on 01/06/2014).

[62] Lars Aake Fredlund. “A framework for reasoning about Erlang code”. PhD thesis. Tekniska högsk., 2001. URL: http://soda.swedish-ict.se/3113/1/SICS_diss_29.pdf (visited on 01/06/2014).