|  |
| --- |
| TRANSILVANIA University, BRASOV |
| CLOUD HASKELL |
| Haskell support for the development of distributed applications |
|  |
| **Alexandra Flinta** |
| **George Abordioaie** |
| **11/1/2016** |

|  |
| --- |
|  |

Table of contents

[1. Abstract 3](#_Toc440230374)

[2. Use of functional programming - Haskell 3](#_Toc440230375)

[2.1. Why use Haskell? 3](#_Toc440230376)

[2.2. What's good about functional programming? 3](#_Toc440230377)

[3. Distributed Applications 4](#_Toc440230378)

[4. Cloud Haskell Platform 6](#_Toc440230379)

[4.1. Asynchronous message passing 6](#_Toc440230380)

[4.2. Copying semantics 6](#_Toc440230381)

[4.3. Process monitoring 7](#_Toc440230382)

[4.4. Selective message reception 7](#_Toc440230383)

[5. Network Transport Abstraction Layer 9](#_Toc440230384)

[6. Concurrency and Distribution 10](#_Toc440230385)

[7. What is Serializable? 11](#_Toc440230386)

[8. Typed Channels 11](#_Toc440230387)

[8.1. Linking and monitoring 12](#_Toc440230388)

[8.2. Stopping Processes 12](#_Toc440230389)

[9. Rethinking the Task Layer 13](#_Toc440230390)

[9.1. Managed Processes 15](#_Toc440230391)

[10. Building Examples 17](#_Toc440230392)

[10.1. Getting Started 17](#_Toc440230393)

[10.2. Building the libraries 18](#_Toc440230394)

[10.3. Setting up the project 19](#_Toc440230395)

[10.4. Create a node 19](#_Toc440230396)

[10.5. Sending messages 20](#_Toc440230397)

[10.6. Serializable data 22](#_Toc440230398)

[10.7. Spawning Remote Processes 22](#_Toc440230399)

[10.8. Classic Ping Pong Example 24](#_Toc440230400)

[11. Final conclusions 27](#_Toc440230401)

[Bibliography 28](#_Toc440230402)

Table of figures

[Figure 1- Distributed application services 4](#_Toc440230403)

[Figure 2 - Dependencies in Cloud Haskell application 8](#_Toc440230404)

[Figure 3 - Cloud Haskell Modules 8](#_Toc440230405)

[Figure 4 - Use of NetworkTransport API 10](#_Toc440230406)

[Figure 5 - Node Creation 10](#_Toc440230407)

[Figure 6 - Primitives for Stating a Process 11](#_Toc440230408)

[Figure 7 - Create channel by Call/Send/Receive Chan primitives 12](#_Toc440230409)

[Figure 8 - Asynchronous Application Example 14](#_Toc440230410)

[Figure 9 - Asynchronously performed task 15](#_Toc440230411)

[Figure 10 - Process implemented with ManagedProcess 16](#_Toc440230412)

[Figure 11 - Cabal Update 18](#_Toc440230413)

[Figure 12 - Cabal sandox init 18](#_Toc440230414)

[Figure 13 - Cabal install 18](#_Toc440230415)

[Figure 14 - Cabal install other variations 18](#_Toc440230416)

[Figure 15 - runhaskell Setup 19](#_Toc440230417)

[Figure 16 - Cabal commands 19](#_Toc440230418)

[Figure 17 - stack new 19](#_Toc440230419)

[Figure 18 - Library import into Main.hs 19](#_Toc440230420)

[Figure 19 - Set IP and port for Network Transport Backend 20](#_Toc440230421)

[Figure 20 - Start new process 20](#_Toc440230422)

[Figure 21 - Message sending application demo 21](#_Toc440230423)

[Figure 22 - Serializable type class 22](#_Toc440230424)

[Figure 23 - Use of Typable instance 22](#_Toc440230425)

[Figure 24 - Use of mkClosure 23](#_Toc440230426)

[Figure 25 - Configuration of remote table 23](#_Toc440230427)

[Figure 26 - Use of sample task 23](#_Toc440230428)

[Figure 27 - Generation of a remote call table 24](#_Toc440230429)

[Figure 28 - Running the code 24](#_Toc440230430)

[Figure 29 - Ping Pong algorithm 25](#_Toc440230431)

[Figure 30 - Ping Pong algorithm print 25](#_Toc440230432)

# Abstract

The following paper tries to familiarize the reader with the basic concepts of distributed applications, concurrency and the improvements brought by Haskell to this branch of computer science. Over ten chapters has been presented a birds eye view of Haskell and functional programming as a whole, concurrency and distributed applications and have dived into the building and usage of various libraries of *Cloud Haskell Platform* which offers open source implementations of distributed computing interfaces, allowing processes to communicate with one other through explicit message passing rather than shared memory.

# Use of functional programming - Haskell

Haskell is a computer programming language (Jones, 2013). In particular, it is a polymorphically statically typed, lazy, purely functional language, quite different from most other programming languages. The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on the lambda calculus, hence the lambda we use as a logo.

## Why use Haskell?

Writing large software systems, that work, is difficult and expensive. Maintaining those systems is even more difficult and expensive. Functional programming languages, such as Haskell, can make it easier and cheaper.

Haskell offers:

* Substantially increased programmer productivity (Ericsson measured an improvement factor of between 9 and 25 using Erlang, a functional programming language similar to Haskell, in one set of experiments on telephony software).
* Shorter, clearer, and more maintainable code.
* Fewer errors, higher reliability.
* A smaller "semantic gap" between the programmer and the language.
* Shorter lead times.

Haskell is a wide-spectrum language, suitable for a variety of applications. It is particularly suitable for programs which need to be highly modifiable and maintainable.

Much of a software product's life is spent in *specification*, *design* and *maintenance*, and not in *programming*. Functional languages are superb for writing specifications which can actually be executed (and hence tested and debugged). Such a specification then *is* the first prototype of the final program.

Functional programs are also relatively easy to maintain, because the code is shorter, clearer, and the rigorous control of side effects eliminates a huge class of unforeseen interactions.

## What's good about functional programming?

Spreadsheets and SQL are both fairly specialized languages. Functional programming languages take the same ideas and move them into the realm of general-purpose programming. To get an idea of what a functional program is like, and the expressiveness of functional languages, look at the following quicksort programs. They both sort a sequence of numbers into ascending order using a standard method called "quicksort". The first program is written in Haskell and the second in C.

Whereas the C program describes the particular steps the machine must make to perform a sort -- with most code dealing with the low-level details of data manipulation -- the Haskell program encodes the sorting algorithm at a much higher level, with improved brevity and clarity as a result (at the cost of efficiency unless compiled by a very smart compiler.

# Distributed Applications

Distributed applications (distributed apps) are applications or software that runs on multiple computers within a network at the same time and can be stored on servers or with cloud computing. Unlike traditional applications that run on a single system, distributed applications run on multiple systems simultaneously for a single task or job.

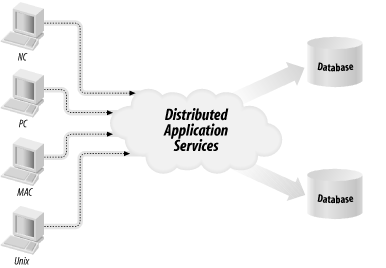


Figure 1- Distributed application services

Distributed apps can communicate with multiple servers or devices on the same network from any geographical location. The distributed nature of the applications refers to data being spread out over more than one computer in a network.

Distributed applications are broken up into two separate programs: the client software and the server software. The client software or computer accesses the data from the server or cloud environment, while the server or cloud processes the data. Cloud computing can be used instead of servers or hardware to process a distributed application's data or programs. If a distributed application component goes down, it can failover to another component to continue running.

Failover is a backup operational mode in which the functions of a system component (such as a processor, server, network, or database, for example) are assumed by secondary system components when the primary component becomes unavailable through either failure or scheduled down time. Used to make systems more fault-tolerant, failover is typically an integral part of mission-critical systems that must be constantly available. The procedure involves automatically offloading tasks to a standby system component so that the procedure is as seamless as possible to the end user. Failover can apply to any aspect of a system: within a personal computer, for example, failover might be a mechanism to protect against a failed processor; within a network, failover can apply to any network component or system of components, such as a connection path, storage device, or Web server.

Distributed applications allow multiple users to access the apps at once. Many developers, IT professionals or enterprises choose to store distributed apps in the cloud because of cloud's elasticity and scalability, as well as its ability to handle large applications or workloads.

# Cloud Haskell Platform

(Cloud Haskell)

Cloud Haskell is a set of libraries that bring *Erlang-style* *concurrency* and distribution to Haskell programs. This project is an implementation of that distributed computing interface, where processes communicate with one another through explicit message passing rather than shared memory.

The term *Erlang-style concurrency* does not have an authoritative definition. Many authors have described their own vision of this term, but the necessary conditions that an application has to fulfill in order to be qualified as an Erland-style application are the following:

* Fast process creation/destruction
* Ability to support >> 10 000 concurrent processes with largely unchanged characteristics.
* Fast asynchronous message passing.
* Copying message-passing semantics (share-nothing concurrency).
* Process monitoring.
* Selective message reception.

## Asynchronous message passing

It has been argued that synchronous message passing is a better building block than asynchronous message passing. It is probably true that synchronous message passing is much easier to reason about, but asynchronous communication (“send and pray”) feels more intuitive in a distributed environment (in which case environments based on synchronous message passing will resort to some form of asynchronous communication as well). Anyway, it is reasonable to argue that asynchronous communication is a defining characteristic of Erlang-style Concurrency.

## Copying semantics

Note that this doesn’t necessarily mean that messages must be copied, but they must act as if they were. There are several reasons why this is important:

* For reliability reasons, processes must not share memory
* In the distributed case, copying is inevitable, and we want as similar semantics as possible between local and distributed message passing.

It is perhaps too strong to require that processes share nothing, as this would rule out Erlang-style concurrency in most existing languages (even languages like Scala make it possible to share data between processes). Let’s just say that the Erlang style is to make it easier to use share-nothing concurrency than to use sharing.

## Process monitoring

This is the enabler of “out-of-band” error handling, or the “let it crash” philosophy, which is very typical for Erlang. The idea is that you program for the correct case, and rely on supervising processes for error handling. This has turned out to be a beautiful way of handling fault tolerance in large systems.

## Selective message reception

There are many ways to achieve selective message reception, but for complex multi-way concurrency, you have to support at least one of them. This is something often overlooked when approaching concurrency. It’s not obviously needed in simple programs, but when it is needed, you will suffer complexity explosion if you don’t have it.

The ability to support selective message reception is sufficient for Erlang-style concurrency. It can be done either by matching on a single message queue (Erlang, OSE Delta), or by using different mailboxes/channels (Haskell, .NET, UNIX sockets).

Originally described by the joint Towards Haskell in the Cloud (Jeff Epstein) paper, Cloud Haskell has be re-written from the ground up and supports a rich and growing number of features for

* building concurrent applications using asynchronous message passing
* building distributed computing applications
* building fault tolerant systems
* running Cloud Haskell nodes on various network transports
* working with several network transport implementations (and more in the pipeline)
* supporting static values (required for remote communication)

Cloud Haskell comprises the following components, some of which are complete, others experimental.

* [distributed-process](https://github.com/haskell-distributed/distributed-process): Base concurrency and distribution support
* [distributed-process-platform](https://github.com/haskell-distributed/distributed-process-platform): The Cloud Haskell Platform - APIs
* [distributed-static](http://hackage.haskell.org/package/distributed-static): Support for static values
* [rank1dynamic](http://hackage.haskell.org/package/rank1dynamic): Like Data.Dynamic and Data.Typeable but supporting polymorphic values
* [network-transport](http://hackage.haskell.org/package/network-transport): Generic Network.Transport API
* [network-transport-tcp](http://hackage.haskell.org/package/network-transport-tcp): TCP realisation of Network.Transport
* [network-transport-inmemory](https://github.com/haskell-distributed/network-transport-inmemory): In-memory realisation of Network.Transport (incomplete)
* [network-transport-composed](https://github.com/haskell-distributed/network-transport-composed): Compose two transports (very preliminary)
* [distributed-process-simplelocalnet](http://hackage.haskell.org/package/distributed-process-simplelocalnet): Simple backend for local networks
* [distributed-process-azure](http://hackage.haskell.org/package/distributed-process-azure): Azure backend for Cloud Haskell (proof of concept)

One of Cloud Haskell’s goals is to separate the transport layer from the process layer, so that the transport backend is entirely independent. In fact other projects can and do reuse the transport layer, even if they don’t use or have their own process layer

Abstracting over the transport layer allows different protocols for message passing, including TCP/IP, UDP, MPI, CCI, ZeroMQ, SSH, MVars, Unix pipes, and more. Each of these transports provides its own implementation of the Network.Transport API and provide a means of creating new connections for use within Control.Distributed.Process.

The following diagram shows dependencies between the various subsystems, in an application using Cloud Haskell, where arrows represent explicit directional dependencies.

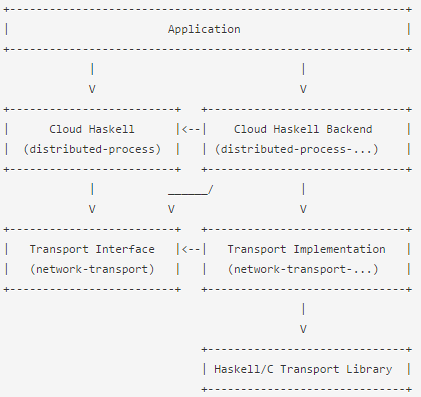


Figure 2 - Dependencies in Cloud Haskell application

In this diagram, the various nodes roughly correspond to specific modules:

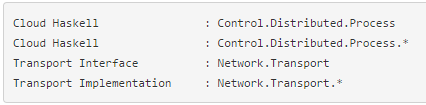


Figure 3 - Cloud Haskell Modules

An application is built using the primitives provided by the Cloud Haskell layer, provided by the Control.Distributed.Process module, which defines abstractions such as nodes and processes.

The application also depends on a Cloud Haskell Backend, which provides functions to allow the initialisation of the transport layer using whatever topology might be appropriate to the application.

It is, of course, possible to create new Cloud Haskell nodes by using a Network Transport Backend such as Network.Transport.TCP directly.

The Cloud Haskell interface and backend make use of the Transport interface provided by the Network.Transport module. This also serves as an interface for the Network.Transport.\* module, which provides a specific implementation for this transport, and may, for example, be based on some external library written in Haskell or C.

# Network Transport Abstraction Layer

Cloud Haskell’s generic network-transport API is entirely independent of the concurrency and messaging passing capabilities of the process layer. Cloud Haskell applications are built using the primitives provided by the process layer (i.e., distributed-process), which provides abstractions such as nodes and processes. Applications must also depend on a Cloud Haskell backend, which provides functions to allow the initialisation of the transport layer using whatever topology might be appropriate to the application.

Network.Transport is a network abstraction layer geared towards specific classes of applications, offering the following high level concepts:

* Nodes in the network are represented by EndPoints. These are heavyweight stateful objects.
* Each EndPoint has an EndPointAddress.
* Connections can be established from one EndPoint to another using the EndPointAddress of the remote end.
* The EndPointAddress can be serialised and sent over the network, whereas EndPoints and connections cannot.
* Connections between EndPoints are unidirectional and lightweight.
* Outgoing messages are sent via a Connection object that represents the sending end of the connection.
* Incoming messages for **all** of the incoming connections on an EndPoint are collected via a shared receive queue.
* In addition to incoming messages, EndPoints are notified of other Events such as new connections or broken connections.

This design was heavily influenced by the design of the Common Communication Interface ([CCI](http://www.olcf.ornl.gov/center-projects/common-communication-interface/)). Important design goals are:

* Connections should be lightweight: it should be no problem to create thousands of connections between endpoints.
* Error handling is explicit: every function declares as part of its type which errors it can return (no exceptions are thrown)
* Error handling is “abstract”: errors that originate from implementation specific problems (such as “no more sockets” in the TCP implementation) get mapped to generic errors (“insufficient resources”) at the Transport level.

For the purposes of most Cloud Haskell applications, it is sufficient to know enough about the Network.Transport API to instantiate a backend with the required configuration and pass the returned opaque handle to the Node API in order to establish a new, connected, running node. More involved setups are, of course, possible. The simplest use of the API is:



Figure 4 - Use of NetworkTransport API

Here we can see that the application depends explicitly on the defaultTCPParameters and createTransport functions from Network.Transport.TCP, but little else. The application can make use of other Network.Transport APIs if required, but for the most part this is irrelevant and the application will interact with Cloud Haskell through the Process Layer and Platform.

# Concurrency and Distribution

The Process Layer is where Cloud Haskell’s support for concurrency and distributed programming are exposed to application developers. This layer deals explicitly with

The core of Cloud Haskell’s concurrency and distribution support resides in the distributed-process library. As well as the APIs necessary for starting nodes and forking processes on them, we find all the basic primitives required to

* spawn processes locally and remotely
* send and receive messages, optionally using typed channels
* monitor and/or link to processes, channels and other nodes

Most of this is easy enough to follow in the haddock documentation and the various tutorials. Here we focus on the essential concepts behind the process layer. A concurrent process is somewhat like a Haskell thread - in fact it is a forkIO thread - but one that can send and receive messages through its process mailbox. Each process can send messages asynchronously to other processes, and can receive messages synchronously from its own mailbox. The conceptual difference between threads and processes is that the latter do not share state, but communicate only via message passing.

Code that is executed in this manner must run in the Process monad. Our process will look like any other monad code, plus we provide an instance of MonadIO for Process, so you can liftIO to make IO actions available.

Processes reside on nodes, which in our implementation map directly to the Control.Distributed.Processes.Node module. Given a configured Network.Transport backend, starting a new node is fairly simple:

4.png

Figure 5 - Node Creation

Once this function returns, the node will be *up and running* and able to interact with other nodes and host processes. It is possible to start more than one node in the same running program, though if you do this they will continue to send messages to one another using the supplied Network.Transport backend.

Given a new node, there are two primitives for starting a new process.

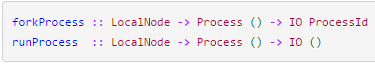


Figure 6 - Primitives for Stating a Process

Once we’ve spawned some processes, they can communicate with one another using the messaging primitives provided by [distributed-processes][distributed-processes], which are well documented in the haddocks.

# What is Serializable?

Processes can send data if the type implements the Serializable typeclass, which is done indirectly by implementing Binary and deriving Typeable. Implementations are already provided for primitives and some commonly used data structures. As programmers, we see the messages in nice high-level form (e.g., Int, String, Ping, Pong, etc), however these data have to be encoded in order to be sent over a communications channel.

Not all types are Serializable, for example concurrency primitives such as MVar and TVar are meaningless outside the context of threads with a shared memory. Cloud Haskell programs remain free to use these constructs within processes or within processes on the same machine though. If you want to pass data between processes using ordinary concurrency primitives such as STM then you’re free to do so. Processes spawned locally can share types such as TMVar just as normal Haskell threads would.

# Typed Channels

Channels provide an alternative to message transmission with send and expect. While send and expect allow us to transmit messages of any Serializable type, channels require a uniform type. Channels work like a distributed equivalent of Haskell’s Control.Concurrent.Chan, however they have distinct ends: a single receiving port and a corollary send port.

Channels provide a nice alternative to bare send and receive, which is a bit un-Haskell-ish, since our process’ message queue can contain messages of multiple types, forcing us to undertake dynamic type checking at runtime.

We create channels with a call to newChan, and send/receive on them using the {send,receive}Chan primitives:

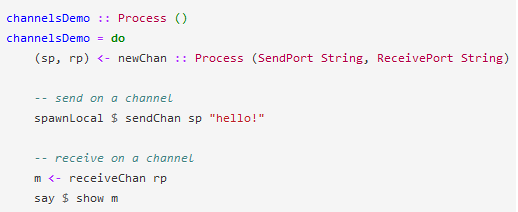


Figure 7 - Create channel by Call/Send/Receive Chan primitives

Channels are particularly useful when you are sending a message that needs a response, because we know exactly where to look for the reply.

Channels can also allow message types to be simplified, as passing a ProcessId for the reply isn’t required. Channels aren’t so useful when we need to spawn a process and send a bunch a messages to it, then wait for replies however; we can’t send a ReceivePort since it is not Serializable.

ReceivePorts can be merged, so we can listen on several simultaneously. In the latest version of distributed-process, we can listen for regular messages and multiple channels at the same time, using matchChan in the list of allowed matches passed receiveWait and receiveTimeout.

## Linking and monitoring

Processes can be linked to other processes, nodes or channels. Links are unidirectional, and guarantee that once the linked object dies, the linked process will also be terminated. Monitors do not cause the listening process to exit, but rather they put a ProcessMonitorNotification into the process’ mailbox. Linking and monitoring are foundational tools for supervising processes, where a top level process manages a set of children, starting, stopping and restarting them as necessary.

## Stopping Processes

Because processes are implemented with forkIO we might be tempted to stop them by throwing an asynchronous exception to the process, but this is almost certainly the wrong thing to do. Firstly, processes might reside on a remote node, in which case throwing an exception is impossible. Secondly, if we send some messages to a process’ mailbox and then dispatch an exception to kill it, there is no guarantee that the subject will receive our message before being terminated by the asynchronous exception.

To terminate a process unconditionally, we use the kill primitive, which dispatches an asynchronous exception (killing the subject) safely, respecting remote calls to processes on disparate nodes and observing message ordering guarantees such that send pid "hello" >> kill pid "goodbye" behaves quite unsurprisingly, delivering the message before the kill signal.

Exit signals come in two flavours however - those that can be caught and those that cannot. Whilst a call to kill results in an un-trappable exception, a call to exit :: (Serializable a) => ProcessId -> a -> Process () will dispatch an exit signal to the specified process that can be caught. These signals are intercepted and handled by the destination process using catchExit, allowing the receiver to match on the Serializable datum tucked away in the exit signal and decide whether to oblige or not.

# Rethinking the Task Layer

Towards Haskell in the Cloud describes a multi-layered architecture, in which manipulation of concurrent processes and message passing between them is managed in the process layer, whilst a higher level API described as the task layer provides additional features such as

* automatic recovery from failures
* data centric processing model
* a promise (or future) abstraction, representing the result of a calculation that may or may not have yet completed

The distributed-process-platform library implements parts of the task layer, but takes a very different approach to that described in the original paper and implemented by the remote package. In particular, we diverge from the original design and defer to many of the principles defined by Erlang’s Open Telecom Platform, taking in some well-established Haskell concurrency design patterns along the way.

In fact, distributed-process-platform does not really consider the task layer in great detail. We provide an API comparable to remote’s Promise in Control.Distributed.Process.Platform.Async. This API however, is derived from Simon Marlow’s Control.Concurrent.Async package, and is not limited to blocking queries on Async handles in the same way. Instead our API handles both blocking and non-blocking queries, polling and working with lists of Async handles. We also eschew throwing exceptions to indicate asynchronous task failures, instead handling task and connectivity failures using monitors. Users of the API need only concern themselves with the AsyncResult, which encodes the status and (possibly) outcome of the computation simply.

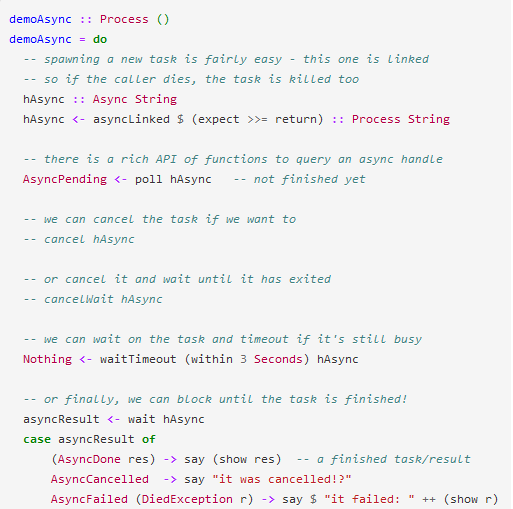


Figure 8 - Asynchronous Application Example

Unlike remote’s task layer, we do not exclude IO, allowing tasks to run in the Process monad and execute arbitrary code. Providing a monadic wrapper around Async that disallows side effects is relatively simple, and we do not consider the presence of side effects a barrier to fault tolerance and automated process restarts. Erlang does not forbid IO in its processes, and yet that doesn’t render supervision trees ineffective. They key is to provide a rich enough API that stateful processes can recognize whether or not they need to provide idempotent initialization routines.

The utility of preventing side effects using the type system is, however, not to be sniffed at. A substrate of the ManagedProcess API is under development that provides a safe process abstraction in which side effect free computations can be embedded, whilst reaping the benefits of the framework.

Work is also underway to provide abstractions for managing asynchronous tasks at a higher level, focusing on workload distribution and load regulation.

The kinds of task that can be performed by the async implementations in distributed-process-platform are limited only by their return type: it **must** be Serializable - that much should’ve been obvious by now. The type of asynchronous task definitions comes in two flavors, one for local nodes which require no remote-table or static serialization dictionary, and another for tasks you wish to execute on remote nodes.

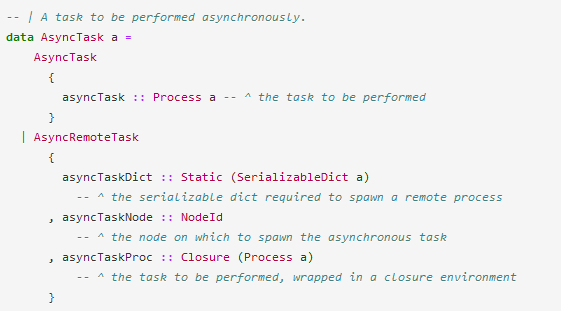


Figure 9 - Asynchronously performed task

## Managed Processes

The main idea behind a ManagedProcess is to separate the functional and non-functional aspects of an actor. By functional, we mean whatever application specific task the actor performs, and by non-functional we mean the concurrency or, more precisely, handling of the process’ mailbox and its interaction with other actors (i.e., clients).

Looking at typed channels, we noted that their insistence on a specific input domain was more haskell-ish than working with bare send and receive primitives. The Async sub-package also provides a type safe interface for receiving data, although it is limited to running a computation and waiting for its result.

The Control.Distributed.Processes.Platform.ManagedProcess API provides a number of different abstractions that can be used to achieve similar benefits in your code. It works by introducing a standard protocol between your process and the world outside, which governs how to handle request/reply processing, exit signals, timeouts, sleeping/hibernation with threadDelay and even provides hooks that terminating processes can use to clean up residual state.

The API documentation is quite extensive, so here we will simply point out the obvious differences. A process implemented with ManagedProcess can present a type safe API to its callers although that’s not its primary benefit. For a very simplified example:

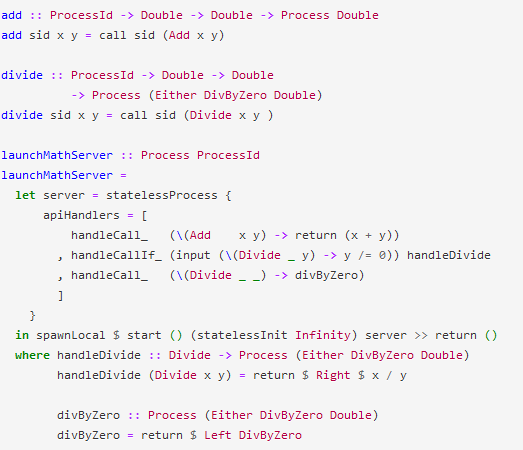


Figure 10 - Process implemented with ManagedProcess

Apart from the types and the imports, that is a complete definition. Whilst it’s not so obvious what’s going on here, the key point is that the invocation of call in the client facing API functions handles all of the relevant waiting/blocking, converting the async result and so on. Note that the managed process does not interact with its mailbox at all, but rather just provides callback functions which take some state and either return a new state and a reply, or just a new state. The process is managed in the sense that its mailbox is under someone else’s control.

# Building Examples

## Getting Started

(Tuyl, 2008)

1. In order to go through these examples, you will need a working Haskell environment. If you don’t already have one, follow the instructions from the Haskell portal in order to install.
2. In order to compile the projects, you will need to install stack. Stack is a cross-platform program for developing Haskell projects. It is aimed at Haskellers both new and experienced.

Stack features:

* Installing GHC automatically, in an isolated location.
* Installing packages needed for your project.
* Building your project.
* Testing your project.
* Benchmarking your project

(Coutts, 2012)

1. To facilitate the use of Cloud Haskell the developers offer a large suite of libraries. These libraries need to be compiled. The user needs to install Cabal in order to use the built libraries.

Cabal is a system for building and packaging Haskell libraries and programs. It defines a common interface for package authors and distributors to easily build their applications in a portable way. Cabal is part of a larger infrastructure for distributing, organizing, and cataloging Haskell libraries and programs.

Specifically, the Cabal describes what a Haskell package is, how these packages interact with the language, and what Haskell implementations must to do to support packages. The Cabal also specifies some infrastructure (code) that makes it easy for tool authors to build and distribute conforming packages.

The Cabal is only one contribution to the larger goal. In particular, the Cabal says nothing about more global issues such as how authors decide where in the module name space their library should live; how users can find a package they want; how orphan packages find new owners; and so on.

(Iubomir, 2015)

1. Once you’re up and running, you’ll want to get hold of the distributed-process library and a choice of network transport backend. This guide will use the network-transport-tcp backend, but other backends are available on Hackage and GitHub.

## Building the libraries

Once the distributed-process and the network-transport-tcp libraries are downloaded, they need to be built. By far the easiest way is to use the cabal command line tool to install a new package and its dependencies.cabal is part of the Haskell Platform, so make sure you install that first.

Once you have the tool installed, installing other packages is easy. The first thing to do is to give the command:



Figure 11 - Cabal Update

This will download the most recent list of packages; this must be done from time to time, to get the latest version of each package, when installing.

It is advisable to use a sandbox, to prevent version incompatibility with earlier installed packages. To initiate a sandbox, give command:



Figure 12 - Cabal sandox init

To install Cabal packages from Hackage use:



Figure 13 - Cabal install

Other common variations:

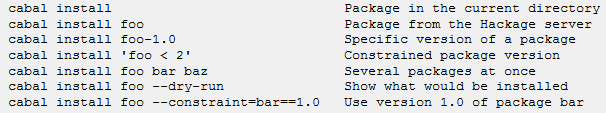


Figure 14 - Cabal install other variations

One thing to be especially aware of, is that the packages are installed locally by default, whereas the commands



Figure 15 - runhaskell Setup

Install globally by default. If you install a package globally, the local packages are ignored. The default for cabal-install can be modified by editing the configuration file.

Help about cabal-install can be obtained by giving commands like:



Figure 16 - Cabal commands

## Setting up the project

Starting a new Cloud Haskell project using stack is as easy as



Figure 17 - stack new

in a fresh new directory. This will populate the directory with a number of files, chiefly stack.yaml and \*.cabal metadata files for the project. You’ll want to add distributed-process and network-transport-tcp to the build-depends stanza of the executable section.

## Create a node

Cloud Haskell’s lightweight processes reside on a “node”, which must be initialized with a network transport implementation and a remote table. The latter is required so that physically separate nodes can identify known objects in the system (such as types and functions) when receiving messages from other nodes. We will look at inter-node communication later, for now it will suffice to pass the default remote table, which defines the built-in types that Cloud Haskell needs at a minimum in order to run.

In app/Main.hs, we start with our imports:

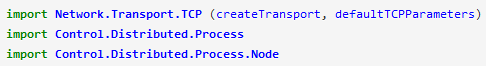


Figure 18 - Library import into Main.hs

Our TCP network transport backend needs an IP address and port to get started with:

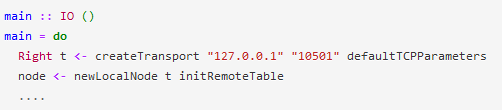


Figure 19 - Set IP and port for Network Transport Backend

And now we have a running node.

## Sending messages

We start a new process by evaluating runProcess, which takes a node and a Process action to run, because our concurrent code will run in the Process monad. Each process has an identifier associated to it. The process id can be used to send messages to the running process - here we will send one to ourselves!

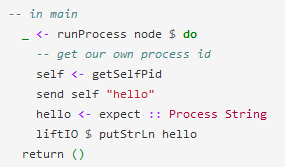


Figure 20 - Start new process

Note that we haven’t deadlocked our own thread by sending to and receiving from its mailbox in this fashion. Sending messages is a completely asynchronous operation - even if the recipient doesn’t exist, no error will be raised and evaluating send will not block the caller, even if the caller is sending messages to itself.

Each process also has a mailbox associated with it. Messages sent to a process are queued in this mailbox. A process can pop a message out of its mailbox using expect or the receive\* family of functions. If no message of the expected type is in the mailbox currently, the process will block until there is. Messages in the mailbox are ordered by time of arrival.

Let’s spawn two processes on the same node and have them talk to each other:

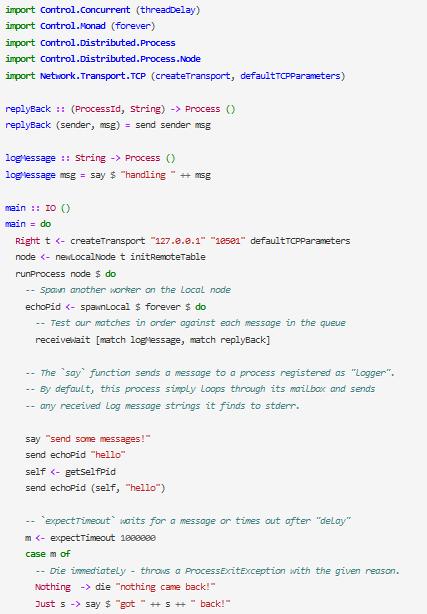


Figure 21 - Message sending application demo

Note that we’ve used receiveWait this time around to get a message. receiveWait and similarly named functions can be used with the Match data type to provide a range of advanced message processing capabilities. The match primitive allows you to construct a “potential message handler” and have it evaluated against received (or incoming) messages. Think of a list of Matches as the distributed equivalent of a pattern match. As with expect, if the mailbox does not contain a message that can be matched, the evaluating process will be blocked until a message arrives which can be matched.

In the echo server above, our first match prints out whatever string it receives. If the first message in our mailbox is not a String, then our second match is evaluated. Thus, given a tuple t :: (ProcessId, String), it will send the String component back to the sender’s ProcessId. If neither match succeeds, the echo server block until another message arrives and tries again.

## Serializable data

Processes may send any datum whose type implements the Serializable type class, defined as:



Figure 22 - Serializable type class

That is, any type that is Binary and Typeable is Serializable. This is the case for most of Cloud Haskell’s primitive types as well as many standard data types. For custom data types, the Typeable instance is always given by the compiler, and the Binary instance can be auto-generated too in most cases, e.g.:

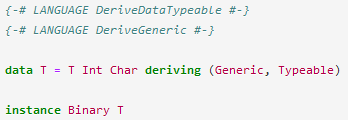


Figure 23 - Use of Typable instance

## Spawning Remote Processes

We saw above that the behaviour of processes is determined by an action in the Process monad. However, actions in the Process monad, no more serializable than actions in the IO monad. If we can’t serialize actions, then how can we spawn processes on remote nodes?

The solution is to consider only static actions and compositions thereof. A static action is always defined using a closed expression (intuitively, an expression that could in principle be evaluated at compile-time since it does not depend on any runtime arguments). The type of static actions in Cloud Haskell is Closure (Process a). More generally, a value of type Closure b is a value that was constructed explicitly as the composition of symbolic pointers and serializable values.

Values of type Closure b are serializable, even if values of type b might not. For instance, while we can’t in general send actions of type Process (), we can construct a value of type Closure (Process ()) instead, containing a symbolic name for the action, and send that instead. So long as the remote end understands the same meaning for the symbolic name, this works just as well. A remote spawn then, takes a static action and sends that across the wire to the remote node.

Static actions are not easy to construct by hand, but fortunately Cloud Haskell provides a little bit of Template Haskell to help. If f :: T1 -> T2 then:



Figure 24 - Use of mkClosure

You can turn any top-level unary function into a Closure using mkClosure. For curried functions, you’ll need to uncurry them first (i.e. “tuple up” the arguments). However, to ensure that the remote side can adequately interpret the resulting Closure, you’ll need to add a mapping in a so-called remote table associating the symbolic name of a function to its value. Processes can only be successfully spawned on remote nodes of all these remote nodes have the same remote table as the local one.

We need to configure our remote table (see the API reference for more details) and the easiest way to do this, is to let the library generate the relevant code for us.

For example:

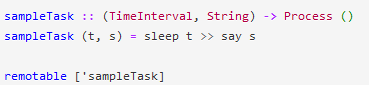


Figure 25 - Configuration of remote table

The last line is a top-level Template Haskell splice. At the call site for spawn, we can construct a Closure corresponding to an application of sampleTask like so:



Figure 26 - Use of sample task

The call to remotable implicitly generates a remote table by inserting a top-level definition \_\_remoteTable :: RemoteTable -> RemoteTable in our module for us. We compose this with other remote tables in order to come up with a final, merged remote table for all modules in our program:



Figure 27 - Generation of a remote call table

In the above example, we spawn sampleTask on node us in two different ways:

* using spawn, which expects some node identifier to spawn a process on along for the action of the process
* using spawnLocal, a specialization of spawn to the case when the node identifier actually refers to the local node (i.e. us). In this special case, no serialization is necessary, so passing an action directly rather than a Closure works just fine

Running the above code will result in printing of the data exchange between the server and the client:

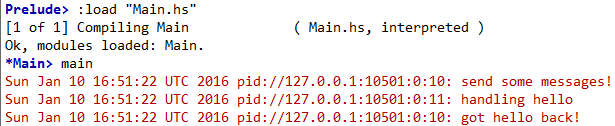


Figure 28 - Running the code

## Classic Ping Pong Example

The above example does nothing more than printing the request and response from client/server. The next example, based on the first one, respects the so called “ping-pong” principle.

We assume that there are two threads *A* and *B*. The thread *A* sends a text (in our example, but in real life applications this may vary from a variable to a very large and complex request), let’s say *Ping* and pauses himself after that. Another thread *B* waits for this text, ‘awakes’ himself, sends the text *Pong* and then goes back to ‘sleep’ allowing thread *A* to restart this process.

This thread synchronization procedure, it may seem without reason but it’s a very important and basic example of concurrency and must be understood by every developer that wishes to approach more complex and large algorithms:

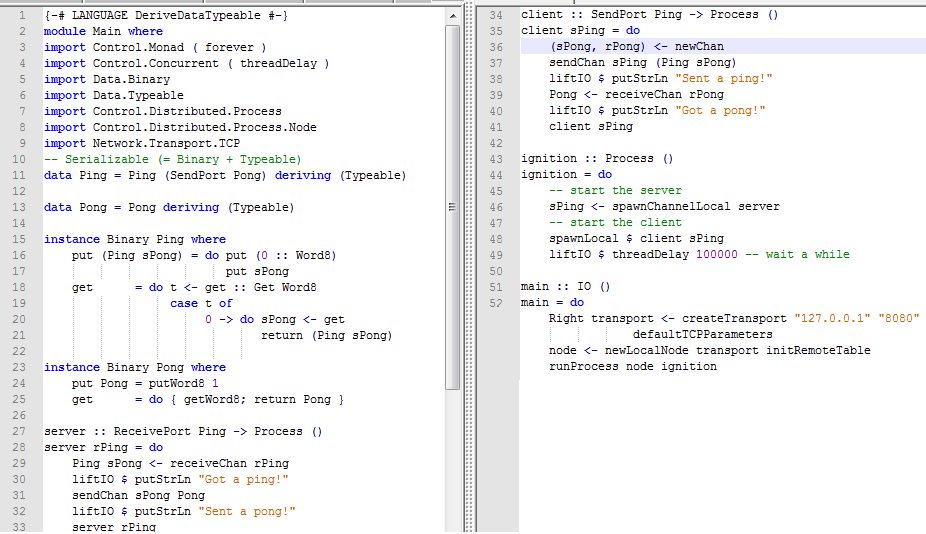


Figure 29 - Ping Pong algorithm

Running the above algorithm will result in a synchronized print of *Got a ping!* and *Sent a pong!* messages:

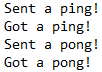


Figure 30 - Ping Pong algorithm print

Due to the speed of Haksell computation and the relative slow speed of message printing, sometimes, this messages will be scrambled, proving the computing speed and efficiency of this Haskell algorithm.

# Final conclusions

Understanding these two basic examples can make every Haskell developer or enthusiast understand, or at least get a grasp of what concurrency is like under this procedural language. *Cloud Haskell Platform* offers complete libraries that support concurrency and proves once again the computing power of Haskell and its versatility. Adding to this statement, other open source packages like *stack* or *cabal* are important tools to know and have for every Haskell developer.

In conclusion, this paper can serve every Haskell enthusiast in their effort to familiarize with the suite offered by *Cloud Haskell Platform*, with the tools used for building various libraries for Haskell or with the theoretical aspects of concurrency and distributed programming.

# Bibliography

*Cloud Haskell*. (n.d.). Retrieved 01 2016, from Cloud Haskell Platform: http://haskell-distributed.github.io/documentation.html

Coutts, D. (2012, 1 28). *Sneezy.cs*. Retrieved 01 2016, from Cloud Haskell: http://sneezy.cs.nott.ac.uk/fun/2012-02/coutts-2012-02-28.pdf

Iubomir. (2015, 11). *GITHUB*. Retrieved 01 2016, from CommercialHaskell: https://github.com/commercialhaskell/stack/tree/master/doc

Jeff Epstein, A. P.-J. (n.d.). *Microsoft Research*. Retrieved 01 2016, from Towards Haskell in the Cloud: http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf

Jones, B. o. (2013, 10 08). *Wiki Haskell*. Retrieved 01 2016, from The Haskell Programming Language: https://wiki.haskell.org/Introduction

Tuyl, H.-J. v. (2008, 11 30). *Wiki Haskell*. Retrieved 01 2016, from Cabal-Install: https://wiki.haskell.org/Cabal-Install