Advanced Functional Programming
Cloud Haskell Chat Tutorial

Peter Eriksson
Kristian Sallberg

June 9, 2014

1 Introduction

In this tutorial we will learn how to use Cloud Haskell. Cloud Haskell is a Haskell approach to the kind of actor based concurrency programming made popular by Erlang. It is highly inspired by Erlang and we will thus begin by comparing them. It is an EDSL (embedded domain specific language) designed to help developers distribute computations. The embedding is shallow, which means most of the work happens close to the semantics as opposed to a deep embedding in which most of the work happens in run functions.

This will hopefully get us up to speed with Cloud Haskell. We then move on to a more advanced use case where we create a chat server and a chat client node. And we show how to spawn some client nodes and connect them to the server.

Readers of this tutorial are expected to understand basic Erlang constructs such as sending and receiving messages, spawning processes and passing around process ids.

1.1 Erlang; dynamic and untyped

Due to increasing usage of concurrency and distribution in new software, programming languages such as Erlang has recently gained popularity. Erlang is a functional language (although not purely functional) designed for developing programs that can be distributed among several computers on a network. Erlang uses asynchronous message passing to communicate between processes.

One of the biggest drawbacks with Erlang is that it's dynamically typed. It means that the compiler won't help us poor developers type check code and it leads to a higher risk of run time errors. We want to explore the Cloud Haskell package (which is an Erlang implementation in Haskell) to see the advantages and disadvantages of the concurrency concept in a strongly typed environment.

2 Message passing

Erlang's concurrency model is based around actors. In Erlang's virtual machine, it's possible to create many nodes. Each node can contain many processes (or actors). Actors communicate through asynchronous message passing, and each actor keeps it's internal state completely inaccessible to the outside world, except for when asked something as part of a message. This means the concurrency model avoids the need of communicating through shared memory, and thus avoids the need of locks, semaphores and all the problems they drag along.

2.1 Example; the actor model and message passing.

We will now look at how message passing actually works in Erlang. Save the following code as message$_{p}assing.erl$:

```
-module(message_passing).
-compile(export_all).
```

We are defining a tail recursive function that will be run inside of an actor. It uses the receive keyword to listen for messages. In the first option if the receive construct, another node is passed on. We can send something to that node using the exclamation mark syntax as shown below.

```
nodeA() ->
    receive
        {ask,Node} -> Node ! {ask,self()};
        {answer,Msg} -> io:fwrite(Msg)
    end,
    nodeA().
```

We create a similar recursive function that also listens for messages using the receive syntax. All it does is responding to an incoming message with a string.

```
nodeB() ->
    receive
        {ask,Node} -> Node ! {answer,"Msg from B!"}
    end,
    nodeB().
```

Make sure to navigate your terminal window to the folder where MessagePassing.erl is located and start the interactive erlang mode by typing *erl*. By following the instructions beneath, line by line, we compile the module; spawn a couple of processes (or actors) and make them interact. Node that the spawn function has arity three in the example, it's possible to spawn processes in other ways too.

We send a message to node A, passing a tuple with the atom ask and a reference to node B. Node A is listening for this, and sends a new message to node B. Node B is listening for this message, and replies to node A with answer ”Msg from B!”. Node A listens for this and prints it to the console. Go ahead and try!

```
c(MessagePassing).
NA = spawn(MessagePassing,nodeA,[]).
NB = spawn(MessagePassing,nodeB,[]).
NA ! {ask,NB}.
```

3 Cloud Haskell version

3.1 Serializable = Binary + Typeable

As we’ve seen, in Erlang it’s possible to send atoms and tuples as they are and the system will convert everything to an appropriate format for transporting data. Cloud Haskell requires us to provide an instance of Binary (functions get and put) so that data types to send and receive can be properly encoded and decoded from binary representation. Cloud Haskell also wants us to derive the type class Typeable for all our data types that will be send around by the transport layer.

3.2 Cloud Haskell message passing

Let’s install Cloud Haskell using Cabal. In a terminal window, type the following commands:

```
cabal update
cabal install cabal-install
cabal distributed-process
cabal distributed-process-simplelocalnet
```

This will update your cabal packages and will install Cloud Haskell, together with a simple Cloud Haskell networking backend that we use in this tutorial.

Save the following code example as CloudExample.hs, then type runhaskell CloudExample.hs in your terminal.

```
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Typeable
import Data.Binary

import Control.Monad
import Control.Distributed.Process
import Control.Distributed.Process.Node

import Network.Transport.TCP
```

We import Data.Typeable and Data.Binary. They are two type classes used for encoding Haskell data types into binary a format suitable for sending over a network connection. Because we are using the language extension DeriveDataTypable, we do not need to create an instance of Typable for our data type SenderPacket, it happens automatically.

However, we have to create an instance of the Binary type class. Put instructs Haskell how to encode the data type to binary data; a zero will be followed by the encoded String and ProcessId. Get discards the initial zero and then decodes the String and ProcessId again. (skriva om lift?)

```
data SenderPacket = SenderPacket String ProcessId
    deriving (Typeable,Show)

instance Binary SenderPacket where
    put (SenderPacket s i) = put (0::Word8) >> put s >> put i
    get                    = do _ <- getWord8 -- stripping 0
                                liftM2 SenderPacket get get
```

We have to create a socket to be able to communicate over a network connection. In Cloud Haskell, this is done by calling createTransport and proving a host IP and a port.

The createTransport might fail, for instance if the provided port number isn't available. In this case, a pattern match of transportToEval will end up at (Left a) and print a failure message to the terminal.

If everything works, we will create a local Cloud Haskell node that will be used to host two processes/actors. Their actions are described below this piece of code. But let's begin by studying it!

```
main :: IO ()
main = do
    transportToEval <- createTransport "127.0.0.1" "11001" defaultTCPParameters
    case transportToEval of
        (Right a) -> do
            env <- newLocalNode a initRemoteTable
            procA <- forkProcess env
                (do liftIO $ putStrLn "procA: Wait for message."
                    (SenderPacket str id) <- expect :: Process SenderPacket
                    send id "gotcha!"
                    liftIO $ putStrLn str
                )
            procB <- forkProcess env
                (do pickOut <- getSelfPid
                    send procA (SenderPacket ("hello!") pickOut)
                    receivedMsg <- expect :: Process String
                    liftIO $ putStrLn $ "got:" ++ receivedMsg
                )
            closeLocalNode env

        (Left a) ->
            putStrLn $ "Fail: " ++ show a
```

The result is spawning two processes; procA and procB. The first process waits to be contacted by the second process, and will write the message received to the terminal and then send "gotcha" back to procB. procB starts out with sending a message to procA and then waits for a response, which is printed afterwards.

LiftIO is used to bring the input output actions up from the Process monad to the IO monad that is defined as an inner monad.

As we can see, a drawback with Cloud Haskell is more lines of code. Erlang is a lot shorter. This is a trade off would you think dynamically typed languages are too unsafe to use.

3.3 Process Monad

The Process monad is the core of Cloud Haskell, many constructor functions return objects wrapped in the Process monad. This makes it possible to process and sequence distributed computations. In fact, we have already used the Process monad. In the following code taken from the simple example above, we are indeed doing the process communication inside the Process monad.

```
do  liftIO $ putStrLn "procA: Wait for message."
    (SenderPacket str id) <- expect :: Process SenderPacket
    send id "gotcha!"
    liftIO $ putStrLn str
```

The process monad data type is a newtype which is more optimized than a simple data keyword. It is an extension of the ReaderT transformer. It stores the information about connected processes in a record format. The Process monad also derives some other construcs, such as Applicative, Monad and MonadIO.

It is implemented as follows.

```
-- | The Cloud Haskell 'Process'
typenewtype Process a = Process {unProcess :: ReaderT LocalProcess IO a}
    deriving (Functor, Monad, MonadIO, MonadReader LocalProcess, Typeable, Applicative)
```

4 Tutorial — Chat server and Clients

We have now learned enough to create a multi process master/slave application, we will create a chat server!

This example uses the SimpleLocalnet module which offers automatic node discovery. Using this module simplifies our work since we don't have to dig deep into configurations and instead instantly gets us going with Cloud Haskell.

Save the following code as Chat.hs

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveDataTypeable #-}
```

```
import System.Environment (getArgs)
import Control.Monad
import Control.Distributed.Process
import Control.Distributed.Process.Closure
import Control.Distributed.Process.Backend.SimpleLocalnet
import Control.Distributed.Process.Node (initRemoteTable,runProcess,forkProcess)

import Data.Typeable
import Data.Binary
```

We have to create a data type representing the type of info we want to send between the server and the clients. The server needs to inform the clients about its process id and since this is a chat, we also want to send messages.

The ChatData data type will represent this, and as in the example in the introduction, the data type must be an instance of typeable and binary.

```
data ChatData = MasterInfo ProcessId | ChatMessage String
    deriving (Typeable, Show)

instance Binary ChatData where
    put (MasterInfo p)  = put (0::Word8) >> put p
    put (ChatMessage s) = put (1::Word8) >> put s
    get                 = do val <- getWord8
                             case val of
                                0 -> liftM MasterInfo get
                                1 -> liftM ChatMessage get
```

The following server loop acts as the chat server (or the master process). It's using the SimpleLocalnet module to look for slaves (slaves are registered with a tag and are thus identifiable). Then a process id is generated and its handle is bound to the keyword pid.

All available slaves (chat clients) are iterated through and sent the master's process id. The server then waits for messages from any slave. It waits for ten milliseconds and then starts all over. It expects a message that it can send to all chat clients.

```
masterLoop :: Backend -> [NodeId] -> Process()
masterLoop b _ = forever $
    do slaves <- findSlaves b
       pid <- getSelfPid
       -- Frequently send master pid to possibly new clients
       forM_ slaves (\x -> send x (MasterInfo pid))
       -- Listen for messages, forward each message to all clients
       msg <- expectTimeout 10 :: Process (Maybe ChatData)
       case msg of
            Nothing  -> return ()
            Just msg -> do forM_ slaves (\c -> send c msg)
                           liftIO $ putStrLn (show msg)
```

A slave node, or chat client, is pretty straight forward to set up. All it needs is a function to keep looping, which we pass on as a higher order function.

```
startCustomSlave :: Backend -> Process()-> IO ()
startCustomSlave backend func = do
        node <- newLocalNode backend
        runProcess node func
```

The client loop itself is perhaps a bit more advanced. However, we are still able to follow basically the same flow throughout the function as we would in Erlang.

We again (just like in the master loop) have to generate a process id. This time however, we also register the process with the name "slaveController". This makes it possible to find slaves from the master process. Then we try to receive information about the master. We can spawn a new thread that makes it possible to read input from the user with the getLine function, at the same time as messages are received from the server.

```
slaveLoop :: Process ()
slaveLoop = do
    pid <- getSelfPid
    register "slaveController" pid
    m <- expect :: Process ChatData
    case m of
        (MasterInfo p) -> do
            liftIO $ putStrLn $ "Master pid: " ++ show p
            -- Spawn a process that listens for input
            spawnLocal (sendMsg p)
            -- Listen for messages
            forever ( do msg <- expect :: Process ChatData
                         case msg of
                                 (ChatMessage s) ->
                                     liftIO $ putStrLn s
                                 _ ->
                                     return ()
                    )
        _ -> do liftIO $ putStrLn "error: could not find server"

sendMsg :: ProcessId -> Process ()
sendMsg pid = forever $ do fromUser <- (liftIO getLine)
                           send pid (ChatMessage fromUser)
```

The main function looks for a parameter that tells this program if a master node (chat server) or a slave node (chat client) should be created.

```
main :: IO ()
main = do
    args <- getArgs
    case args of
        ["server", host, port] -> do
            backend <- initializeBackend host port initRemoteTable
            startMaster backend (masterLoop backend)
        ["client", host, port] -> do
            backend <- initializeBackend host port initRemoteTable
            startCustomSlave backend slaveLoop
```

Go ahead and try test the application by writing:

```
runhaskell Chat.hs server 127.0.0.1 10001
runhaskell Chat.hs client 127.0.0.1 10002
runhaskell Chat.hs client 127.0.0.1 10003
runhaskell Chat.hs client 127.0.0.1 10004
```

It launches the chat server, and three clients. You can now (given that the port numbers are available) start chatting between the three chat clients. Happy times!