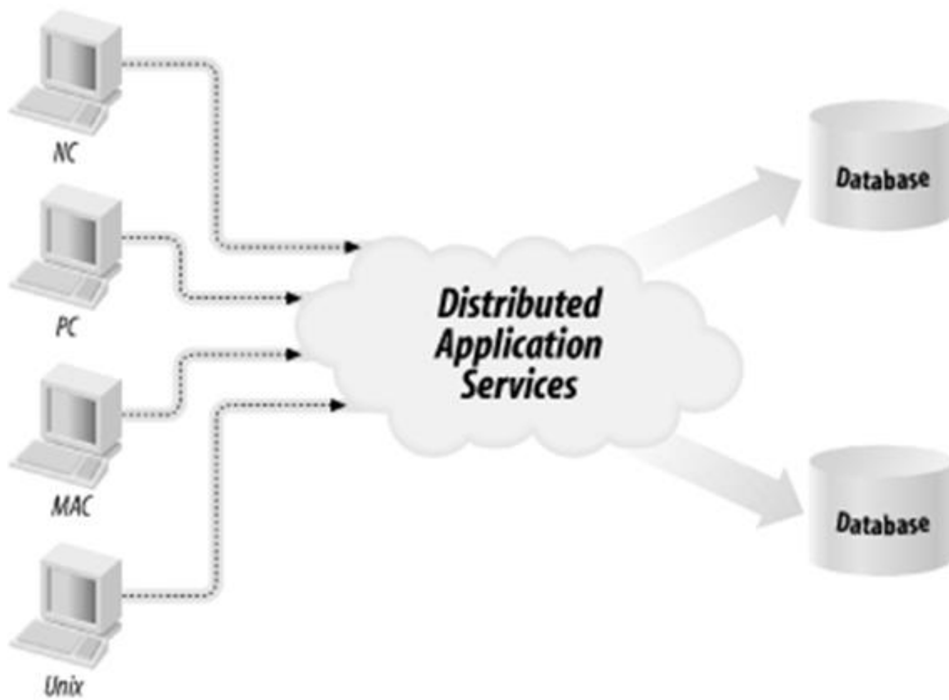
CLOUD HASKELL
Haskell support for the development of distributed applications

Alexandra Flinta
George Abordioaie

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.



Distributed applications

- Can communicate with multiple servers or devices on the same network from any geographical location
- 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
  - The server or cloud processes the data
- It can **failover** to another component If a distributed application component goes down

Distributed applications - benefits

- Allow multiple users to access the apps at once
- Developers, IT professionals or enterprises choose to store distributed apps in the cloud:
  - cloud elasticity
  - cloud scalability
  - ability to handle large applications
  - ability to handle workloads

Cloud Haskell – Erlan-style concurrency

- Cloud Haskell is a set of libraries that bring Erlang-style concurrency and distribution to Haskell programs
  - 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.

Cloud Haskell – New approach

- 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 – Components

- Cloud Haskell comprises the following components, some of which are complete, others experimental.
  - distributed-process: Base concurrency and distribution support
  - distributed-process-platform: The Cloud Haskell Platform - APIs
  - distributed-static: Support for static values
  - rank1dynamic: Like Data.Dynamic and Data.Typeable but supporting polymorphic values
  - network-transport: Generic Network.Transport API
  - network-transport-tcp: TCP realisation of Network.Transport
  - network-transport-inmemory: In-memory realisation of Network.Transport (incomplete)
  - network-transport-composed: Compose two transports (very preliminary)
  - distributed-process-simplelocalnet: Simple backend for local networks
  - distributed-process-azure: Azure backend for Cloud Haskell (proof of concept)

Cloud Haskell – Support for Haskell

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

```
+------------------------------------------------------------+
|                        Application                         |
+------------------------------------------------------------+
             |                                |
             V                                V
+-------------------------+   +----------------------------+
|      Cloud Haskell      |<--|    Cloud Haskell Backend   |
|  (distributed-process)  |   | (distributed-process-...)  |
+-------------------------+   +----------------------------+
             |           ______/              |
             V           V                    V
+-------------------------+   +----------------------------+
|   Transport Interface   |<--|   Transport Implementation |
|   (network-transport)   |   |   (network-transport-...)  |
+-------------------------+   +----------------------------+
                                              |
                                              V
                              +----------------------------+
                              | Haskell/C Transport Library |
                              +----------------------------+
```

Cloud Haskell – Support for Haskell

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

```
Cloud Haskell                : Control.Distributed.Process
Cloud Haskell                : Control.Distributed.Process.*
Transport Interface          : Network.Transport
Transport Implementation     : Network.Transport.*
```

- *Control.Distributed.Process* module defines abstractions such as nodes and processes
- The Transport interface provided by the *Network.Transport* module is used by the Cloud Haskell interface and backend
- The *Network.Transport.** module provides a specific implementation for the current transport

Concurrency and distributed applications

- The *Process Layer* is where Cloud Haskell's support for concurrency and distributed programming are exposed to application developers.
- 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:

```
newLocalNode :: Transport -> IO LocalNode
```

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

```
forkProcess :: LocalNode -> Process () -> IO ProcessId
runProcess  :: LocalNode -> Process () -> IO ()
```

Building Examples

- Prerequisites:
  - Haskell environment
  - Stack: cross-platform program for developing Haskell projects
  - Cabal: system for building and packaging Haskell libraries
  - `distributed-process` and `network-transport-tcp` libraries from Cloud Haskell Platform

Building the libraries

- Download and install Cabal
- Download and unzip `distributed-process` and `network-transport-tcp` libraries
- Go to the root folder of every library and type: `cabal install library-name.cabal`

Setting up the project

- Setting up the project is made using stack

- Type `stack new project-name` 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.

```
import Network.Transport.TCP (createTransport, defaultTCPParameters)
import Control.Distributed.Process
import Control.Distributed.Process.Node
```

```
main :: IO ()
main = do
  Right t <- createTransport "127.0.0.1" "10501" defaultTCPParameters
  node <- newLocalNode t initRemoteTable
  ....
```

Sending Messages

- We start a process by evaluating `runProcess` which takes a `Process` and a node to run:

```
-- in main
  _ <- runProcess node $ do
    -- get our own process id
    self <- getSelfPid
    send self "hello"
    hello <- expect :: Process String
    liftIO $ putStrLn hello
  return ()
```

- Each process has an identifier associated to it.
- Each process also has a mailbox associated with it. Messages sent to a process are queued in this mailbox in a asynchronous .

Sending Messages II

- `receiveWait` and similarly named functions can be used with the `Match` data type to provide a range of advanced message processing capabilities.

```
-- Test our matches in order against each message in the queue
receiveWait [match logMessage, match replyBack]
```

Sending Messages III

- In the `echo server`, 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.

```
-- Spawn another worker on the local node
echoPid <- spawnLocal $ forever $ do
```

```
replyBack :: (ProcessId, String) -> Process ()
replyBack (sender, msg) = send sender msg
```

Serializable Data

- Processes may send any type of data as long as they implement the `Serializable` typeclass:

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

- Any type that is `Bynary` and `Typable` is `Serializable`.
- For custom data types a Typable instance is given by the compiler and a Bynary instance can be generated as such:

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveGeneric #-}

data T = T Int Char deriving (Generic, Typeable)

instance Binary T
```

Spawning Remote Processes

- 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?

- Solution? – Static actions …

Spawning Remote Processes II

- 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:

```
$(mkClosure 'f) :: T1 -> Closure T2
```

- You can turn any top-level unary function into a `Closure` using `mkClosure`
- For curried functions you need to move the arguments into tuples and make a mapping into the remote table.

```
sampleTask :: (TimeInterval, String) -> Process ()
sampleTask (t, s) = sleep t >> say s

remotable ['sampleTask]
```

Spawning Remote Processes III

- 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.