The New Cloud Haskell

Duncan Coutts and Edsko de Vries

September 2012, Haskell Implementors Workshop
What I want to talk about today...

- Very quick recap on Cloud Haskell
- The cool new stuff
  - details of the new implementation
  - message semantics
  - current status

Sorry, not a tutorial
(but come to the Haskell Exchange in London next month!)
Cloud Haskell recap
What’s it all about?

- Slogan could be “Erlang for Haskell” (as a library)
- Concurrent **distributed** programming in Haskell
- A programming model + an implementation
What’s the point?

- To let you program a cluster as a whole,
- or a data centre,
- or a bunch of VMs rented from Azure / Amazon / ... (hence the “Cloud” marketing buzzword)
What’s the point?

- To let you program a cluster as a whole,
- or a data centre,
- or a bunch of VMs rented from Azure / Amazon / ...
  (hence the “Cloud” marketing buzzword)

Key idea

Program the cluster as a whole, not individual nodes
Other people’s good ideas

Papers

- Jeff Epstein, Andrew Black and Simon Peyton Jones, *Towards Haskell in the Cloud*, Haskell Symposium 2011
- Jeff Epstein, *Functional programming for the data centre*, MPhil thesis, 2011

Prototype

- remote package by Jeff Epstein
Programming model

- Explicit concurrency
- Lightweight processes
- No state shared between processes
- Asynchronous message passing

Some people call this the “actor model”
The Cloud Haskell design

Basic approach

- Design is implementable as a library
  - minimal language and RTS changes
  - e.g. no distributed MVar as in GdH
- If in doubt, do it the way Erlang does it

(Other distributed middleware designs are also possible)
The 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
```
Errors are everywhere in distributed programming

Cloud Haskell steals Erlang’s solution

- Let processes fail
  - communication loss counts as failure
- Notify interested processes
  - often they just fail too (linked processes)
  - common pattern is to monitor and restart

link :: ProcessId → Process ()
monitor :: ProcessId → Process MonitorRef
What we’ve been up to...
A new implementation

Simon PJ asked us to start work on a new implementation...

Initial goals

- Same public API (more or less)
- Robust implementation
- Flexible implementation

Interesting problems we ran into

- The need for semantics (!)
- Network disconnect and reconnect
The need for flexibility

Variation between use cases

<table>
<thead>
<tr>
<th>Network data transport layer (hardware and protocol)</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to start your executable on each machine</td>
<td>exotic non-IP HPC networks</td>
</tr>
<tr>
<td>How to configure each node</td>
<td>shared memory or local pipes</td>
</tr>
<tr>
<td>How to find initial peers or all peers</td>
<td>remote login via ssh</td>
</tr>
<tr>
<td></td>
<td>cloud service API</td>
</tr>
<tr>
<td></td>
<td>cluster job scheduler</td>
</tr>
<tr>
<td></td>
<td>via ssh from master node</td>
</tr>
<tr>
<td></td>
<td>config files, env vars, string and glue</td>
</tr>
<tr>
<td></td>
<td>distributed via cluster job scheduler</td>
</tr>
<tr>
<td></td>
<td>discover dynamically on LAN</td>
</tr>
<tr>
<td></td>
<td>known from config</td>
</tr>
<tr>
<td></td>
<td>cluster job scheduler</td>
</tr>
<tr>
<td></td>
<td>peers created in new VMs</td>
</tr>
</tbody>
</table>
The new implementation

Key differences with the prototype implementation

- Swappable network transport layer
- Multiple Cloud Haskell backends to handle
  - selection of transport implementation
  - initialisation
  - configuration
  - peer discovery / creation
- More precisely specified semantics
  - message passing
  - node disconnect and reconnect
Existing prototype design

Your distributed application

Cloud Haskell

standard network library (TCP/IP)

libraries of distributed algorithms & patterns
Network transport layer

Interface between network layer and Process layer

- Allows different network implementations
- Clarifies internal design of Cloud Haskell

Design considerations

- Meet needs of Cloud Haskell
- Be reusable in other projects if possible
- Allow many implementations with common semantics
- Allow high performance (latency)
- Allow high scalability (big clusters)
Network transport layer

Key features

- heavyweight endpoints
- bundle of many lightweight connections between endpoints
- connections are
  - message oriented (not stream)
  - reliable and ordered (like TCP)
  - unidirectional
- single shared receive queue on each endpoint
  - all incoming messages from all connections
  - errors and other events
- clear network failure behaviour
  - explicit reporting of failures
  - bundles fail as a whole, not individual connections
Network transport layer

Implementations

- TCP/IP
  - multiplexes lightweight connections over a single heavyweight TCP connection between endpoints
- Unix pipes (in progress)
- CCI (in progress)
  (CCI is an HPC networking lib supporting infiniband etc)

Also possible

- Shared memory
- SSH
- UDP
- TCP with SSL/TLS

The TCP implementation is already being used in projects other than Cloud Haskell
Process layer outline

- Cloud Haskell node manages a set of processes
  - transport Endpoint per node
- Each Process runs in a Haskell thread
  - has a queue for incoming messages
- A lightweight transport Connection per pair of communicating processes
- A thread per node to receive events
  - dispatches messages to per-process message queues
  - passes messages and notifications to the node controller
  - handles network error events (like peer node disconnect)
- A thread per node as the “node controller”
  - responsible for spawning, linking and monitoring
  - also manages a process registry (named processes)
- Other per-node service processes
  - currently just a logger
Cloud Haskell backends

“SimpleLocalnet” backend

- simple backend to get started quickly
- no configuration
- uses the TCP transport
- node discovery using local UDP multicast
Cloud Haskell backends

Windows Azure backend

- uses Linux VMs
- uses the TCP transport between the VMs
- initialise with Azure account and SSL certificates
- Support for:
  - VM enumeration
  - copying binaries to VMs
  - spawn nodes on VMs
- special API required for communicating between on-cloud and off-cloud nodes
- not yet released
Semantics, semantics, semantics!
Process layer semantics

We started implementing the process layer...

What is the behaviour supposed to be?
What is the spec exactly?

▶ original paper says the message passing is "asynchronous, reliable, and buffered" but little more.

For example, what does this do? `link p; send p "hi!"; unlink p`.

▶ does the link happen before the send?

▶ does the unlink guarantee the message was delivered?

▶ are the link operations sync or async?

▶ any reliability guarantee on message delivery?
Process layer semantics

We started implementing the process layer...

What is the behaviour supposed to be?
What is the spec exactly?

- original paper says the message passing is "asynchronous, reliable, and buffered" but little more
Process layer semantics

We started implementing the process layer...

What is the behaviour supposed to be?
What is the spec exactly?

- original paper says the message passing is “asynchronous, reliable, and buffered” but little more

For example, what does this do?

```plaintext
do link p; send p "hi!"; unlink p
```

- does the `link` happen before the `send`?
- does the `unlink` guarantee the message was delivered?
- are the `link` operations sync or async?
- any reliability guarantee on message delivery?
Process layer semantics

Remember? "If in doubt, do it the way Erlang does it."
Remember? "If in doubt, do it the way Erlang does it."

What is the Erlang spec exactly?

- most of the docs are fuzzy
- but a few good papers which reveal the gory details
Remember? "If in doubt, do it the way Erlang does it."

What is the Erlang spec exactly?

- most of the docs are fuzzy
- but a few good papers which reveal the gory details

The important questions

- behaviour of message passing between two processes?
- behaviour of linking and monitoring?
Message passing guarantees

Meaning of “reliable ordered” message delivery

Process A sends messages to process B:

\[ m_1, m_2, m_3, \ldots \]

Process B may receive any **prefix**.

For example receiving \( m_1, m_3 \) cannot happen.

The Erlang FAQ says

“if you think TCP guarantees delivery, which most people probably do, then so does Erlang”
Message passing guarantees

Meaning of “reliable ordered” message delivery

Process A sends messages to process B:

\( m_1, m_2, m_3, ... \)

Process B may receive any prefix.

For example receiving \( m_1, m_3 \) cannot happen.

The Erlang FAQ says

“if you think TCP guarantees delivery, which most people probably do, then so does Erlang”

But it turns out Erlang does not guarantee this.

Process B can receive just \( m_1, m_3 \)
Erlang formal semantics guarantees ordered messaging between pairs of processes.

It does not guarantee reliable delivery: intermediate messages can be dropped.
Erlang semantics

Erlang formal semantics guarantees ordered messaging between pairs of processes.

It does not guarantee reliable delivery: intermediate messages can be dropped.

In practice dropping messages is rare but can happen when Erlang nodes are disconnected and reconnected.
Proposed future Erlang semantics

We found a good paper:

- Svensson et al. *A unified semantics for future Erlang*, Erlang workshop 2010

They propose what they think Erlang semantics *should* be

- formal specification
- *does* guarantee reliable ordered message delivery
- simplified linking and monitoring
- everything is asynchronous
- covers node disconnect and reconnect (mostly)

We took this as the spec for our implementation.
Proposed future Erlang semantics

We found a good paper:

- Svensson et al. *A unified semantics for future Erlang, Erlang workshop 2010*

They propose what they think Erlang semantics *should* be

- formal specification
- *does* guarantee reliable ordered message delivery
- simplified linking and monitoring
- everything is asynchronous
- covers node disconnect and reconnect (mostly)

We took this as the spec for our implementation.

*If in doubt, do it the way Erlang does it*

*the Erlang people now think Erlang *ought* to do it*
Revisiting the example

So what does this do now?

do link p; send p "hi!"; unlink p

- all asynchronous
- link is **not** ordered wrt. send
- so this code guarantees almost nothing
Revisiting the example

So what does does this do now?

```plaintext
do link p; send p "hi!"; unlink p
```

- all asynchronous
- link is not ordered wrt. send
- so this code guarantees almost nothing

What we probably want instead is

```plaintext
do link p; send p "hi!"; reply ← expect; unlink p
```

- order of link vs send does not matter here

Lessons

- linking has very little to do with message delivery
- to assure delivery you must receive a reply
Question

Why does Erlang not provide reliable delivery when TCP does?

TCP is connection oriented

- you establish a connection to an address and send data over the connection
- network failure is reflected as the connection closing

Erlang (and Cloud Haskell) are connectionless

- you send messages direct to addresses (ProcessIds)

If we allow node reconnects it is hard to mix reliable delivery and connectionless style
Node disconnect and reconnect

Example

Process A sends messages to process B: \( m_1, m_2, m_3, \ldots \)

Now the network between A and B fails. What should we do?

The nodes may be disconnected temporarily or permanently.
Node disconnect and reconnect

Example

Process A sends messages to process B: \( m_1, m_2, m_3, \ldots \)

Now the network between A and B fails. What should we do?

The nodes may be disconnected temporarily or permanently

A few options

- buffer messages
- drop messages temporarily
- drop messages permanently (do not allow reconnect)
Node disconnect and reconnect

Current Erlang behaviour

- buffers messages temporarily
- then drops messages
- sacrifices reliability property

“Unified semantics for future Erlang”

- drops messages to dead nodes
- buffers messages to disconnected nodes
- keeps reliability property
- impossible to implement
Node disconnect and reconnect

Our proposal for Cloud Haskell

- drop messages permanently (by default)
- this keeps the reliability property (!!)
- explicit `reconnect` primitive
- `reconnect` to accept intermediate message loss

We think this is a reasonable compromise

- simple reliability guarantee
- most code does not need to handle reconnect
  - it simply fails on the initial disconnect
- code that wants to handle reconnect explicitly opts in and accepts the reality of message loss
Implementation status
Current state of the implementation

Current status

- Covers the full API
- Made a first release and several minor bug-fix releases
- Reasonable test suite
- Reasonable performance

Ready for serious experiments, but not yet for serious use.
Current state of the implementation

Significant TODOs

- Larger scale testing
- Node disconnect and reconnect needs more work and testing
- More demos
- Comparative benchmarking needed

Wishlist

- Shared memory transport
- SSH transport
- Ability to use multiple transports
- Implementation of the ‘static’ language extension
- Higher level libraries, e.g. Erlang OTP’s gen_server

Contributions welcome
Early benchmarks

Transport layer microbenchmark of the TCP implementation

- minimal overhead compared to network package
- some latency overhead compared to C
  - primarily issues in the threaded RTS and GHC I/O manager

Process layer microbenchmark comparison with the prototype

- approx 4x lower latency
- approx 200x greater throughput

(-running on Azure infrastructure)

This is not a surprising result:
the prototype uses synchronous message send

Benchmarking against Erlang is required
Cloud Haskell Packages

Cloud Haskell Packages on Hackage

distributed-process  Main API, Process etc
distributed-process-simplelocalnet Simple backend
distributed-process-azure Windows Azure backend

network-transport Transport interface
network-transport-tcp TCP implementation

Sources and documentation on github
http://github.com/haskell-distributed/distributed-process
Thanks!

Questions?
Extra slides
Initialisation sequence looks something like

```haskell
import Control.Distributed.Process
import Network.Transport.TCP

init :: (...) -> Process () -> IO ()
init config initialProcess = do
  transport ← createTransport config
  localnode ← newLocalNode transport
  runProcess localnode initialProcess
```

- initialise a transport, with some transport-specific config
- initialise the local Cloud Haskell node
- run the initial process

This is all hidden in a Cloud Haskell backend
newtype Ping = Ping ProcessId deriving (Binary, Typeable)
ping :: Process ()
ping = do 
  self ← getSelfPid
  Ping partner ← expect
  send partner (Ping self)
  say "ping!"

initialProcess _ = do 
  nid ← getSelfNode
  ping1 ← spawn nid ping__closure
  ping2 ← spawn nid ping__closure
  send ping1 (Ping ping2)

$(remotable ['ping]) -- Template Haskell magic

main = remoteInit (Just "config") [__remoteCallMetaData]
initialProcess
Asyncronous primitives

\[
\text{spawn} :: \text{NodeId} \rightarrow \text{Closure (Process())} \rightarrow \text{Process ProcessId}
\]

\[
\text{spawn \ nid \ proc} = \text{do}
\]

\[
\text{us} \leftarrow \text{getSelfPid};
\]

\[
\text{mRef} \leftarrow \text{monitorNode \ nid}
\]

\[
\text{sRef} \leftarrow \text{spawnAsync \ nid \ (childClosure \ proc)}
\]

\[
\text{mPid} \leftarrow \text{receiveWait}
\]

\[
\begin{align*}
[&\text{matchIf} \\
&\quad (\lambda (\text{DidSpawn \ ref \ _}) \rightarrow \text{ref} \equiv \text{sRef}) \\
&\quad (\lambda (\text{DidSpawn \ _ \ pid}) \rightarrow \text{return} \ (\text{Right \ pid}))
\end{align*}
\]

\[
\begin{align*}
, &\text{matchIf} \\
&\quad (\lambda (\text{NodeMonitorNotification \ ref \ _ \ _}) \rightarrow \text{ref} \equiv \text{mRef}) \\
&\quad (\lambda (\text{NodeMonitorNotification \ _ \ _ \ err}) \rightarrow \text{return} \ (\text{Left \ err}))
\end{align*}
\]

\[
\text{unmonitor} \ \text{mRef}
\]

\[
\text{case \ mPid \ of}
\]

\[
\begin{align*}
&\text{Left \ _err} \rightarrow \text{return} \ (\text{nullProcessId \ nid}) \\
&\text{Right \ pid} \rightarrow \text{send} \ \text{pid} () \Rightarrow \text{return} \ \text{pid}
\end{align*}
\]