Einführung in Cloud Haskell

Stefan Schmidt

10. Januar 2013

Inhalt

Einführung

Überblick

Cloud Haskell

- Slogan: “Erlang for Haskell (as a library)”
- Bibliothek zur Erstellung von verteilten, nebenläufigen Systemen
- Message Passing
- Remote Process spawning

Entwickelt von

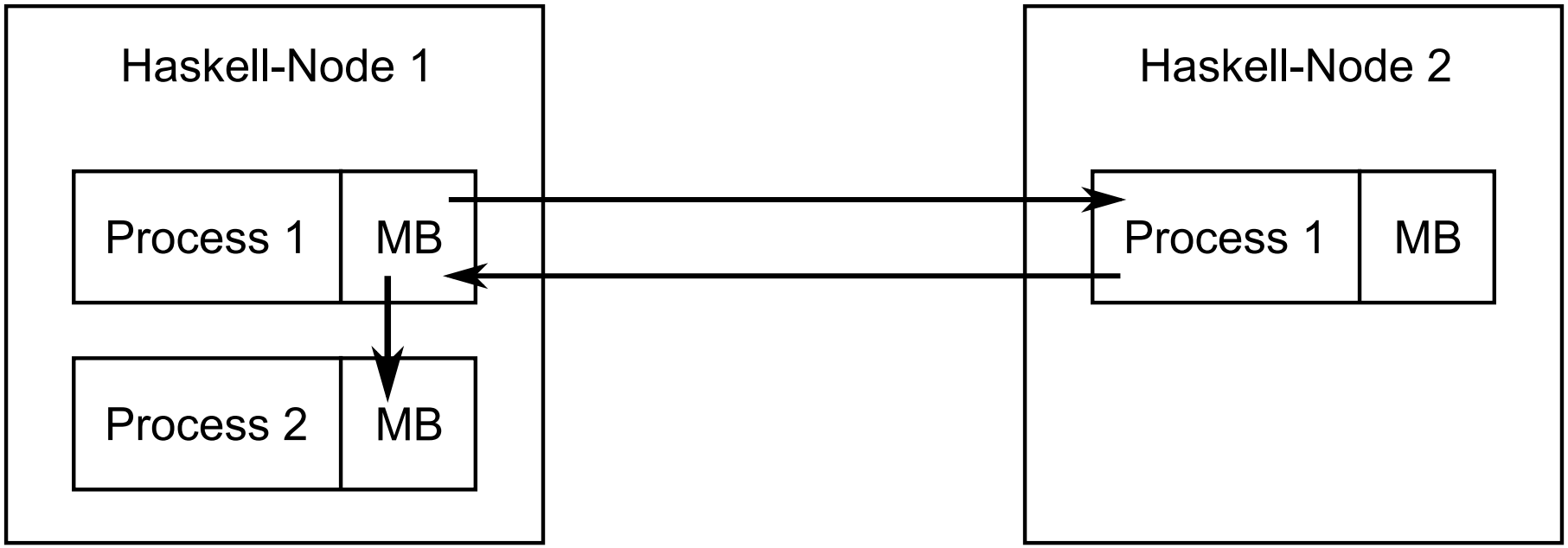
- Duncan Coutts
- Nicolas Wu
- Edsko de Vries

Überblick

- Haskell Programm bildet einen “Haskell Node”
- innerhalb eines Nodes können mehrere Prozesse parallel Abaufen
- explizite Nebenläufigkeit
- leichtgewichtige Prozesse
- Prozesse haben keinen geteilten Zustand
- jeder Process hat eigene Mailbox
- asynchrones Message Passing, auch zu Prozessen auf anderen Nodes

$\Rightarrow$ “Actor Modell”

Überblick


Haskell-Node 1
Process 1
MB
Process 2
MB
Haskell-Node 2
Process 1
MB

Cloud Haskell Packages auf Hackage

Main API

- `distributed-process`
- `network-transport` Transport interface
- `network-transport-tcp` TCP implementation

Backends

- `distributed-process-simplelocalnet`
- `distributed-process-azure`

Github:
`http://github.com/haskell-distributed/distributed-process`

Entwicklung

Paper

- 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

Prototyp

- remote package by Jeff Epstein

Message Passing und lokale Prozesse

Message Passing API

```
data Process a
instance Monad Process
instance MonadIO Process

data ProcessId
data NodeId

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

-- Send a message
send :: Serializable a => ProcessId -> a -> Process ()

-- Wait for a message
expect :: forall a. Serializable a => Process a

-- Our own process ID
getSelfPid :: Process ProcessId

-- The node ID of our local node
getSelfNode :: Process NodeId
```

Erzeugen von lokalen Prozessen

Erzeugen eines neuen lokalen Prozesses:

```
-- Spawn a process on the local node from another process
spawnLocal :: Process () -> Process ProcessId
```

Wie wird der erste Prozess erzeugt?

```
-- Run a process on a local node and wait for it to finish
runProcess :: LocalNode -> Process () -> IO ()
```

Wie wird der lokale Node erzeugt?

```
-- initialize a new local node.
newLocalNode :: Transport -> RemoteTable -> IO LocalNode
```

- Transport - zu benutzender Transport Layer
- RemoteTable - “öffentliche Schnittstelle des Node”

Manuelle Erzeugung eines Node

Manuelle Erzeugung eines Node,
der über TCP mit anderen Nodes kommuniziert:

```
import Network.Transport.TCP as TCP
import Control.Distributed.Process.Node

startLocalNode :: String -> String -> Process () -> IO ()
startLocalNode host port initialProcess
  = do
    mbTransport <- TCP.createTransport host port TCP.
        defaultTCPParameters
    case mbTransport of
      (Left e) -> error $ show e
      (Right transport) -> do
        localnode <- newLocalNode transport initRemoteTable
        runProcess localnode initialProcess
```

Demo: spawning local processes

Demo: spawning local processes

Verarbeitung unterschiedlicher Nachrichten-Typen

- Prozess Mailbox akzeptiert alle Nachrichten-Typen
- `expect` liefert nur die erste Nachricht des jeweils gewünschten Typs
- alle anderen Nachrichten verbleiben in der Mailbox (wie in Erlang)

Lösung:

- Angabe einer Liste von Handler-Funktionen
- Jede Handler-Funktion ist für die Verarbeitung eines Nachrichtentyps zuständig
- solange über Mailbox iterieren und dabei testen, ob für den jeweiligen Nachrichtentyp eine Handler-Funtion angegeben wurde

Verarbeitung unterschiedlicher Nachrichten-Typen

```
data Match b

-- Test the matches in order against each message in the
    queue
receiveWait :: [Match b] -> Process b

-- Match against any message of the right type
match :: forall a b. Serializable a => (a -> Process b) ->
    Match b

-- Match against any message of the right type that
    satisfies a predicate
matchIf :: forall a b. Serializable a => (a -> Bool) -> (a
    -> Process b) -> Match b

-- Remove any message from the queue
matchUnknown :: Process b -> Match b
```

Demo: message type matching

Demo: message type matching

Channels

Zusätzlich auch Channel-Objekte:

```
data ReceivePort a
data SendPort a  -- instance of Binary

-- Create a new typed channel
newChan :: Serializable a => Process (SendPort a,
    ReceivePort a)

-- Send a message on a typed channel
sendChan :: Serializable a => SendPort a -> a -> Process ()

-- Wait for a message on a typed channel
receiveChan :: Serializable a => ReceivePort a -> Process a
```

Remote Prozesse und Fehlerbehandlung

Erzeugen entfernter Prozesse

Erzeugen eines neuen entfernten Prozesses:

```
-- Create a new process on a different node
spawn :: NodeId -> Closure (Process ()) -> Process
    ProcessId
```

- `Closure (Process ())` - “serialisierter Funktionsaufruf”
- nur Funktionsname und Parameter
- keine Übertragung von Code
- nur Parameter, die `Serializable` implementieren

Registrieren von extern aufrufbaren Funktionen am lokalen Node:

```
remotable [ 'f, 'g, ... ]   -- Template Haskell

-- creates:
__remoteTable :: RemoteTable -> RemoteTable
```

Erzeugen entfernter Prozesse

Erzeugen des Closure-Objektes:

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

Beispiel:

```
doWork :: (String, String) -> Process ()
doWork = ...

remotable ['doWork]

spawnRemote :: NodeId -> Process ()
spawnRemote nId = do
  _ <- spawn nId ($(mkClosure 'doWork) ("Hi","Ho"))
  return ()

...
newLocalNode transport $ __remoteTable initRemoteTable
...
```

Erzeugen entfernter Prozesse

Achtung! Zyklische Abhängigkeiten in Template Haskell Splices:

```
doWork1 :: (String, String) -> Process ()
doWork1 = ...

doWork2 :: () -> Process ()
doWork2 _
  = do
    _ <- spawn nId ($(mkClosure 'doWork1) ("Hi","Ho"))
    return ()

remotable ['doWork1, 'doWork2]

spawnRemote :: NodeId -> Process ()
spawnRemote nId = do
  _ <- spawn nId ($(mkClosure 'doWork2) ("Hi","Ho"))
  return ()
```

Demo: remote process spawning

Demo: remote process spawning

Fehlerbehandlung

Fehlerbehandlung an Erlang angelehnt:

- keine Exception, wenn senden der Nachrichten nicht erfolgreich
- keine Exception, wenn erzeugen eines entfernten Prozesses nicht möglich
- beim Auftreten von Exceptions wird nur der jeweilige Prozess beendet
- Möglichkeit, entfernte Prozesse oder Nodes zu überwachen

Überwachung entfernter Prozesse:

```
-- throws exception if process is not available
link :: ProcessId -> Process ()

-- puts message in mailbox if process is not available
monitor :: ProcessId -> Process MonitorRef
```

Asynchrone Calls

Aufrufe von `link` sind asynchron:

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

- keine Garantie dass `link` vor `send` aufgerufen wird
- keine Garantie, dass Nachricht wirklich ankommt

Warten auf Bestätigung, dass Nachricht wirklich angekommen ist:

```
do link p; send p "hi!"; reply <- expect; unlink p
```

- sollte in der Zwischenzeit ein Fehler auftreten, greift `link`

Demo: remote process linking

Demo: remote process linking

Transport Backends

Bootstrapping

Probleme bei der Initialierung des Netzwerks:

- Woher wissen die Nodes, wie andere Nodes zu erreichen sind?
- Können Nodes sich nicht gegenseitig finden?
- Konstruktor von `NodeId` ist nicht public (Hack in Demos)

Möglichkeiten:

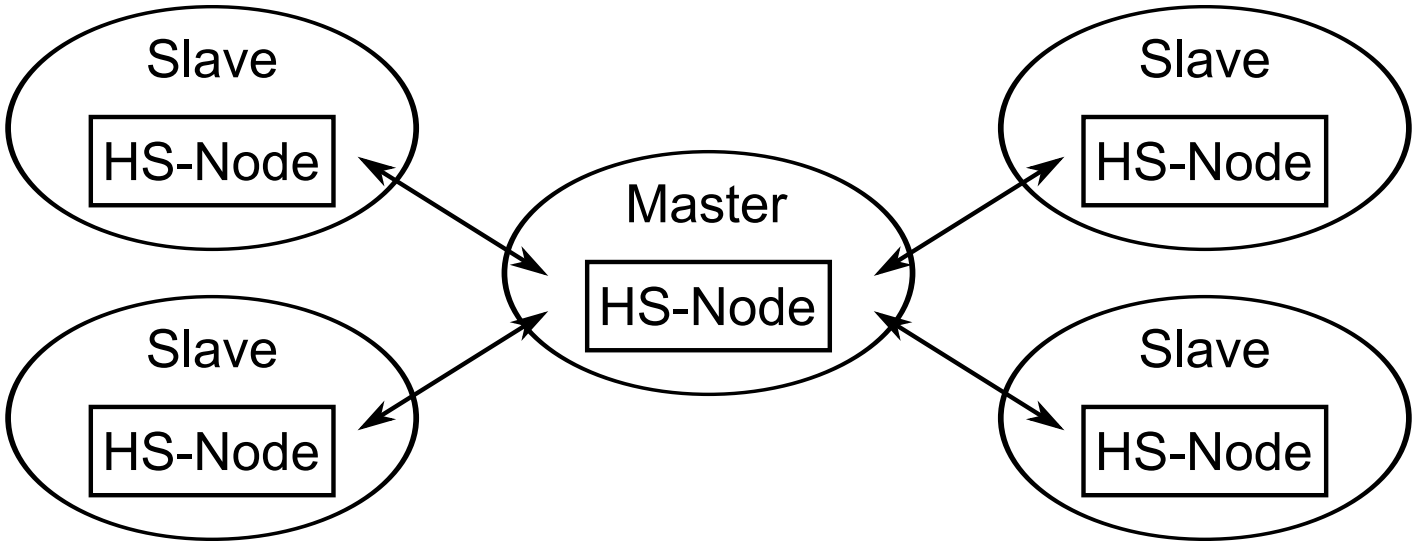
- Low-Level Funktionen zum Nachrichten-Austausch (TCP, UDP)
- `NodeId` ist Serializable
- Nodes haben Registry (Named Processes, Abfrage von außen)

Vorhandene Backends:

- SimpleLocalnet
- Windows Azure

SimpleLocalnet Backend

- Master-Slave-Topographie
- lokales Netzwerk
- wenig Konfiguration
- TCP basiert
- Knoten senden UDP Mulitcast Nachrichten



Demo: SimpleLocalnet

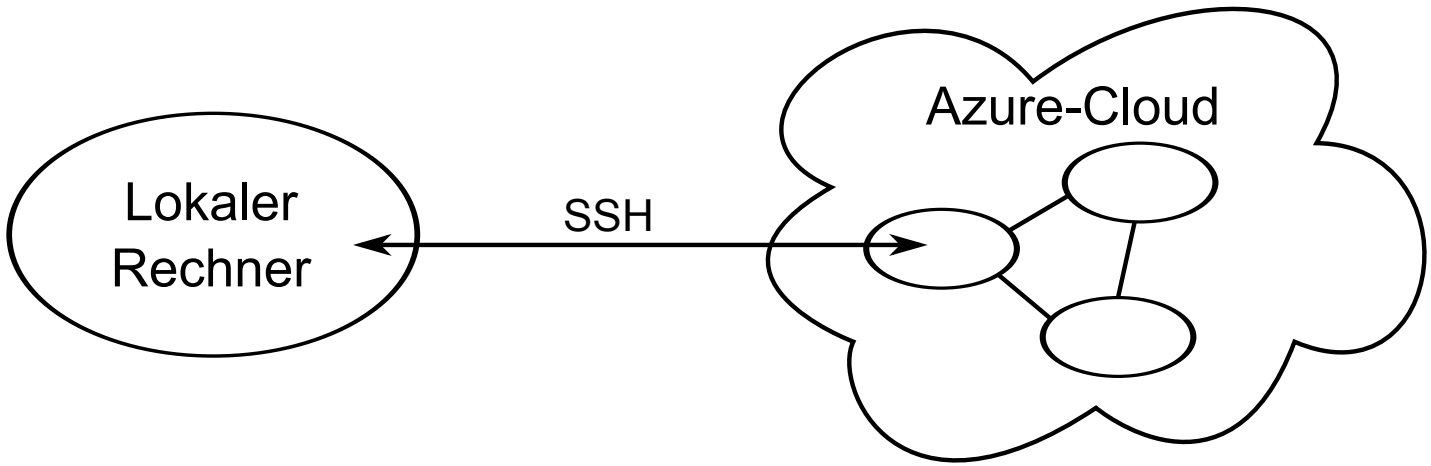
Demo: SimpleLocalnet

Demo: MapReduce mit SimpleLocalnet

Demo: MapReduce mit SimpleLocalnet

Windows Azure Backend

- Controller-Node auf lokalem Rechner
- Worker-Nodes in Cloud
- Kommunikation innerhalb der Cloud: TCP
- Kommunikation zwischen Cloud und lokalem Rechner: SSH
- Linux VMs



Bewertung

Bewertung

- Message Passing API
- Remote Process Spawning
- Error Handling
- Logging
- Benchmarks?
- Geeignet für kommerziellen Einsatz?
- Weitere Entwicklung? (Haskell-OTP?)

Vielen Dank!

Fragen?

Quellen

- Sources und Haddock Documentation auf Hackage
- Duncan Coutts, Cloud Haskell 2.0
  Haskell Implementors Workshop 2012
  `http://www.haskell.org/wikiupload/4/46/Hiw2012-duncan-coutts.pdf`
- Duncan Coutts, Cloud Haskell
  Haskell eXchange 2012
  `http://skillsmatter.com/podcast/home/cloud-haskell`
- Alexander Bondarenko, distributed-process-p2p
  `http://hackage.haskell.org/package/distributed-process-p2p`