

# Functional programming for the data centre

Jeffrey Epstein



University of Cambridge  
Computer Laboratory  
Fitzwilliam College

June 2011

Submitted in partial fulfillment  
of the requirements for the degree of  
Master of Philosophy  
in Advanced Computer Science

## Declaration

I, Jeffrey Epstein of Fitzwilliam College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Some of the ideas presented in this report have appeared in different form in the paper “Haskell for the cloud,” which was written in collaboration with Simon Peyton Jones and Andrew P. Black.

This report contains 14 587 words, including tables, figures, and footnotes, but excluding appendix and bibliography. This count does not exceed the regulation length of 15 000 words.

Signed

Date June 15, 2011

# Functional programming for the data centre

Jeffrey Epstein

## Summary

We present a framework for developing Haskell programs to be run in a distributed computing environment. The framework lets programmers create high-performance, reliable, distributed applications while retaining Haskell's traditional strengths in strong static typing and purity. The framework introduces programmer-level abstractions appropriate to several programming styles and types of applications, including a system for message-passing between concurrent threads; communication by distributed queue; and resolution of data dependency by proxies known as *promises*. We demonstrate the effectiveness of the programming model by presenting several demonstration programs. We also show that an implementation of the  $k$ -means algorithm written using the framework is able to take advantage of a computing cluster environment to improve performance.

## Acknowledgments

This work has been made possible due to the keen insight and extraordinary patience of many people.

I thank Simon Peyton Jones, my supervisor at Microsoft Research, for his excellent help and guidance. I thank Alan Mycroft, my supervisor at the Computer Lab, for his support and advice.

I thank thank Andrew P. Black for his input and invaluable aid.

I thank John Launchbury, who suggested an improvement to the theory of closure serialization.

I thank Koen Claessen, who insightfully pointed out the importance of not making receive ports serializable.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>Design</b>	<b>9</b>
2.1	Messages . . . . .	9
2.1.1	Basic messaging . . . . .	10
2.1.2	Messages and choice . . . . .	12
2.1.3	Messages through channels . . . . .	16
2.2	Processes . . . . .	17
2.3	Fault monitoring . . . . .	19
2.4	Tasks . . . . .	21
2.4.1	Interface . . . . .	21
2.4.2	MapReduce . . . . .	24
<b>3</b>	<b>A practical tour</b>	<b>26</b>
3.1	Code . . . . .	26
3.2	Deployment . . . . .	29
<b>4</b>	<b>Implementation</b>	<b>32</b>
4.1	Message handling . . . . .	32
4.2	System processes . . . . .	33
4.2.1	Spawner . . . . .	33
4.2.2	Process monitor . . . . .	33
4.2.3	Logger . . . . .	35
4.2.4	Discovery . . . . .	36
4.3	Tasks . . . . .	37
4.4	Closures . . . . .	39

---

<b>5</b>	<b>Evaluation</b>	<b>44</b>
5.1	Clustering . . . . .	44
5.2	Performance . . . . .	44
<b>6</b>	<b>Conclusion</b>	<b>48</b>
6.1	Related work . . . . .	48
6.2	Future work . . . . .	50
<b>A</b>	<b>Configuration reference</b>	<b>52</b>

# Chapter 1

## Introduction

With the age of steadily improving processor performance behind us, the way forward seems to be to compute with more, rather than faster, processors. Demanding applications are therefore often run in a *data center*, a network of connected commodity computers, having independent failure modes and separate memories. When a data center is used for storing and processing end users' data or running users' programs, it is termed a *cloud*. Developing software for these distributed computing environments presents some unique challenges:

- There is the obvious need to coordinate program control between many, possibly heterogeneous computers. Most modern, popular programming languages do not directly address distributed concurrency. To the extent that they support concurrency at all, it is usually of a shared-memory variety: that is, it presents the programmer with the abstraction of multiple concurrent threads accessing a pool of mutable common data. This model is not appropriate for a distributed system, where the cost of moving data from one processor to another becomes a dominant factor.
- An application in a distributed environment needs to tolerate failure. When programming a system that spans hundreds or thousands of individual computers, some of them are likely to be failing at any given moment. A failure of a single component should not require restarting the whole calculation, or else it might never finish. At the least, tools for detecting and responding to failure need to be available to the programmer. Ideally, failure would be handled entirely automatically.

In this thesis, we address the above challenges. We present Cloud Haskell, a framework for developing distributed applications in Haskell. The contributions of this report are:

- A proposal for a message-passing distributed programming API, which we call the *process layer* (Chapter 2). Following the Erlang model, the interface provides a

system for exchanging messages between concurrent threads and monitoring them for failure, regardless of whether those threads are running on one computer or on many. We provide a detailed discussion of the implementation of this API (Chapter 4).

- A proposal for a data-centric distributed programming API, which we call the *task layer* (Section 2.4). Inspired by Skywriting, the interface is more abstract than the process layer, and so lets the programmer leave details of fault recovery and data locality to the framework.
- A new method for serializing closures to enable higher-order functions to work in a distributed environment (Section 4.4). The need to start threads remotely demands a representation of code objects and their environment. Our approach to closures requires explicit indication of which parts of the function’s environment will be serialized and thus gives the programmer greater control over his or her application’s cost model.
- A demonstration of the effectiveness of our approach. This demonstration takes two forms: a development walkthrough (Chapter 3) showing how our framework’s programming model lets complex programs be expressed elegantly and deployed easily; and performance measurements of the  $k$ -means clustering algorithm as implemented with our framework (Chapter 5).



# Chapter 2

## Design

We present an overview of Cloud Haskell’s programming interface. This introduction is suitable for programmers wishing to develop distributed applications. A complete set of documentation is available online at <http://www.cl.cam.ac.uk/~jee36/remote/>. We divide our discussion into a presentation of the framework’s *process layer* (Sections 2.1 through 2.3) and its *task layer* (Section 2.4).

Cloud Haskell is built using the Glasgow Haskell Compiler. The source code for the project is available at <http://github.com/jepst/CloudHaskell/>.

### 2.1 Messages

Cloud Haskell’s process layer is a domain-specific language based on the abstractions of *messages* and *processes*. This approach, termed *message passing*, was popularized for high-performance computing by MPI [8] and for real-time applications by Erlang [1]. The message passing model stipulates that concurrent threads have no implicit access to each other’s data, but instead share data explicitly by sending and receiving messages. One great advantage of this model is that it scales easily: you can develop an application on a single computer, and, as fits your needs, you can later redeploy it to a cluster of computers with no design changes. Even better, the message-passing model for thread communication eliminates some of the classic concurrency pitfalls, such as race conditions.

Haskell is an ideal language for this problem space. As a purely functional language, data is by default immutable, so the lack of shared mutable data will not be missed. Also, critically, Haskell’s purity and monads let us control side effects.

The basic unit of concurrency in the process layer is the *process*, a concurrent thread with the ability to send and receive messages. Processes are lightweight, having low overhead to create and schedule, and are identified by a unique process identifier, which can be used to send messages to them. A *message* is a finite, serializable data structure;

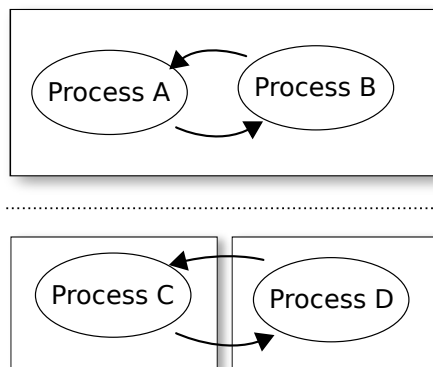


Figure 2.1: Processes A and B run on the same node and therefore share an address space, although they do not share any variables with each other. They communicate exclusively by messages, here represented by arrows. They can be easily reconfigured as Processes C and D, which run on separate nodes.

their transmission is asynchronous, reliable, and buffered. Cloud Haskell’s processes and messages are adapted from the Erlang language, which adheres to a “share-nothing” approach to distributed processes. Erlang has served as the initial inspiration as well as a concrete model for our process layer.

A process needs to run somewhere, and we call that location a *node*. Like processes, each node has a unique identifier. A node is created for every instance of the framework’s runtime. One can imagine a node as representing a single computer, but in practice that is not necessarily the case. In particular, nodes may be deployed on virtual machines, typically as part of a cloud computing infrastructure. Alternatively, a single computer can have multiple nodes. This design makes it possible to design and test distributed applications before deploying to a production system, as shown in Figure 2.1.

Code running in a process executes in the `ProcessM` monad, which keeps track of the state associated with that process, primarily its message queue. Think of the `ProcessM` type as an indication that the function has access to the data structure representing a process, and therefore can pass on this data structure implicitly to other `ProcessM`-monadic functions that it calls, while a function without the `ProcessM` type cannot. Therefore, all functions dealing with the sending and receiving of messages, or the manipulation of processes, are in the `ProcessM` monad.

The `ProcessM` monad is itself based on the `IO` monad, and is in fact an instance of `MonadIO`; therefore, arbitrary `IO` functions can be called from within `ProcessM` code via `liftIO`.

### 2.1.1 Basic messaging

Consider some elementary functions for dealing with messages:

```
send :: (Serializable a) => ProcessId -> a -> ProcessM ()
expect :: (Serializable a) => ProcessM a
```

Before examining the `send` and `expect` functions more closely, consider an example. Take a simple process that accepts a “pong” message and responds by sending a corresponding “ping” to the process that sent it. Using `send` and `expect`, the code for such a process would be:

```
data Ping = Ping ProcessId
data Pong = Pong ProcessId
— omitted: Serializable instances for Ping and Pong
```

```
ping :: ProcessM ()
ping = do { self ← getSelfPid
          ; Pong partner ← expect
          ; send partner (Ping self)
          ; ping }
```

The equivalent code in Erlang looks like this:

```
ping() →
  receive
    {pong, Partner} →
      Partner ! {ping, self()}
  end,
  ping().
```

These two programs have nearly identical structure. Both `ping` functions are designed to be run as processes. The programs look for a “pong” message, and ignore all others. The “pong” message contains in its payload a process ID, to which the response message is sent, this time containing the process ID of the `ping` process (given in both languages as `self`, and in the Haskell version retrieved with a call to the `getSelfPid` function). Finally, they wait for the next message by repeating with tail recursion.

Our general-purpose message-sending primitive, `send`, packages an arbitrary chunk of data and transmits it (possibly over the network) to a process, which we indicate by its `ProcessID`. Upon receipt, the incoming message will be placed in a message queue associated with the destination process. The `send` function corresponds to Erlang’s “!” operator. If `send` cannot deliver the message to the destination process, it will throw an exception, which can be caught with Haskell’s usual exception-handling facilities.

The simplest way of receiving a message is `expect`, which extracts a message from the message queue associated with the current process. It will examine each message in the queue, dequeuing and returning the first message of the correct type; in this case, of the type `Pong`, which is inferred. If no message of the correct type is currently in the queue, `expect` will block until such a message arrives.

The type signatures of `send` and `expect` require that the message must be an instance of the `Serializable` type class. This requirement ensures that the message can be serialized

and transmitted to another process. To be `Serializable`, it is both sufficient and necessary that a data type be an instance of the type classes `Typeable` and `Binary`. `Typeable` lets us query the value’s type at runtime, while `Binary` provides a mechanism for serializing a data structure to a byte stream and reconstructing it from one. These instances are provided for most of Haskell’s types and are easy to extend to user-defined data types: the `Typeable` class can be automatically derived for most user-defined data types, and instantiating `Binary` requires only writing `get` and `put` functions to convert between live and serialized forms of data. In addition, Cloud Haskell includes generic implementations of these functions, `genericPut` and `genericGet`, that work with any instance of `Data`. Here is an example of a serializable user-defined list type:

```
data List a = Cons a (List a) | Nil deriving (Typeable, Data)
instance Binary a  $\Rightarrow$  Binary (List a) where
    put = genericPut
    get = genericGet
```

While all of Haskell’s primitive data types and most of the common higher-level data structures are instances of `Serializable`, and therefore can be part of a message, some are emphatically not. One example is `MVar`, a mutable concurrent variable. Since `MVar` allows communication between threads on the assumption of shared memory, it is not helpful to send it to a process that has no shared memory with the current process. Although one can imagine a synchronous distributed variable that mimics the semantics of an `MVar` (as in Glasgow Distributed Haskell [19]), such a variable would have a vastly different cost model than `MVar`. Since neither `MVar`’s cost model nor its implementation could be preserved in a distributed context, we felt it best to prohibit programmers from trying to use it in that way. Nevertheless, `MVars` can be used within a single process: processes are allowed to use Haskell’s `forkIO` function to create local threads that can share memory using `MVar`. In this, Cloud Haskell differs from Erlang, which has no allowance for shared-memory variables.

### 2.1.2 Messages and choice

In Section 2.1.1, we introduced the `expect` function, which lets us receive messages of a particular type. But what if we want the process to be able to accept messages of multiple types? We would like to be able to approximate the Erlang `receive` syntax:

```
math()  $\rightarrow$ 
  receive
    {add, Pid, Num1, Num2}  $\rightarrow$ 
      Pid ! Num1 + Num2;
    {divide, Pid, Num1, Num2} when Num2  $\neq$  0  $\rightarrow$ 
      Pid ! Num1 / Num2;
    {divide, Pid, _, _}  $\rightarrow$ 
```

```

        Pid ! div_by_zero
    end,
    math().

```

This code accepts and responds to several different messages. It does this by pattern matching on the values of the messages, which are (by convention) all tuples. Different types of messages are differentiated by the atom in the first element of the tuple, which here is either `add` or `divide`. Each pattern is tested in order against each incoming message; when a pattern matches, the message is consumed and the corresponding action is executed, which here sends a response to the originating process.

Haskell lacks an atom data type, but an idiomatic way of representing the different messages is to use type constructors. Our program will accept messages matching both `Add pid a b` and `Divide pid num den`, and will respond by sending a value `Answer Double`, or a `DivByZero` error. It is possible to create a message type that represents the union of all of these variants, suitable for use with `expect`:

```

data MathOp = Add ProcessId Double Double
             | Divide ProcessId Double Double
             | Answer Double
             | DivByZero

```

There are several problems with this approach: Remember that `expect` can select messages only by type. Thus, a process receiving `MathOp` messages would need to be able to respond to all variants, even those that do not make sense: presumably, only client processes should receive `Answer` and `DivByZero` while only the calculating process should receive `Add` and `Divide`. Moreover, putting all messages into a single sum type breaks modularity by exposing details of the calculating process to the client; when we add a new mathematical operation, every client will need to update its code, even if it does not use that operation. Worse, a single process that offers more than one service could not keep them separate; clients of either would be forced to see the interface of both.

To avoid this kind of false dependency, it is better to break the single `MathOp` type into several types, each of which can be handled separately:

```

data AddT = Add ProcessId Double Double
data DivideT = Divide ProcessId Double Double
data DivByZeroT = DivByZero

```

In addition to the above types, the answer can be sent to the client simply as a message of type `Double`—no wrapper required. But now we have a different quandary: although `expect` can receive any type of message, it can receive only one type of message at a time, and will block until a message of that type is put into the message queue. So there is no way to handle the above message types without knowing the order in which the client is

going to send `Adds` and `Divides`. What we need is an alternative to `expect` that provides the notion of *choice*.

To do this, consider how to specify a pair consisting of a message type and its corresponding action. We introduce `match`, which accepts a message handler function as a parameter.

```
match :: (Serializable a) => (a -> ProcessM q) -> MatchM q ()
```

When `match` tests an incoming message, it compares the type of the message against the type `a`, the parameter of the message handler. Any message of that type will be considered “matched”: it will be removed from the message queue, and the message handler will be invoked.

The `match` function is in the `MatchM` monad, which is responsible for providing `match` with the current state of the message queue, and then providing `match`’s caller with the result of its test. The `match` function will typically be used with `receiveWait`, which mimics Erlang’s `receive` syntax by evaluating a list of `MatchMs` in order. The value returned by the selected message action is also the return value of `receiveWait`.

```
receiveWait :: [MatchM q ()] -> ProcessM q
```

We mimic Erlang’s `when` clause, which allows message acceptance to be qualified by a predicate, with `matchIf`, whose first parameter is a function that is allowed to examine the incoming message without removing it from the queue.

```
matchIf :: Serializable a => (a -> Bool) -> (a -> ProcessM q) -> MatchM q ()
```

Now let’s use these tools to implement the `math` function in Haskell:

— *omitted: Serializable instances for AddT, DivideT, and DivByZeroT types*

```
math :: ProcessM ()
math =
  receiveWait
    [ match    (λ(Add pid num1 num2) ->
                send pid (num1 + num2)),
    matchIf (λ(Divide _ _ num2) -> num2 ≠ 0)
        (λ(Divide pid num1 num2) ->
          send pid (num1 / num2)),
    match    (λ(Divide pid _ _) ->
              send pid DivByZero) ]
    >> math
```

The combination of `receiveWait` and `match` closely corresponds to Erlang’s `receive` syntax. The `MatchMs` are tested in order against each message in the message queue. When a matching message is found, the corresponding lambda function is invoked.

Notice that matching by message type is not quite the same as matching by message value. For example, if a particular `match` accepts messages of a certain type, then all variants of that type must be handled. In the above example, this is acceptable, because the `AddT` type has only a single variant, built with the `Add` constructor. If instead it were a sum type with a second constructor, then the `match` call that deals only with the `Add` constructor would raise a pattern match exception if a message with the other constructor were received.

Clearly, `receiveWait` is more flexible than `expect`. In fact, `expect` is implemented in terms of `receiveWait`. Its definition is:

```
expect :: (Serializable a) => ProcessM a
expect = receiveWait [match return]
```

Whether we use `receiveWait` or `expect`, we run the risk that the function will block until a certain type of message arrives. What if we want to check the incoming message queue, but not wait indefinitely? Erlang lets us do this with the `receive ... after` syntax:

```
Pid ! {query, Stuff},
receive
  {response, Answer} →
    show_answer(Answer)
after
  50000 →
    show_error("Timeout!")
end
```

This code will wait at most 50 seconds to receive a response to its query; after 50 seconds, it will stop waiting and display an error message. The corresponding function in Cloud Haskell is `receiveTimeout`. Like `receiveWait`, it takes a list of matches, but it also takes a timeout value. If the timeout is exceeded, the function returns `Nothing`.

```
receiveTimeout :: Int → [MatchM q ()] → ProcessM (Maybe q)
```

Thus we can translate the Erlang example into Haskell:

```
do { send pid (Query stuff)
    ; ret ← receiveTimeout 50000000
      [ match(λ(Response answer) →
              return answer) ]
    ; case ret of
      Nothing → showError "Timeout!"
      Just ans → showAnswer ans }
```

As with Erlang's `receive ... after` syntax, `receiveTimeout` can be called with a timeout value of zero, which has the effect of checking for a matching message and returning immedi-

ately if no match is found. Note that unlike Erlang, `receiveTimeout` takes its argument in microseconds, for consistency with Haskell’s `timeout` function.

### 2.1.3 Messages through channels

As is clear from the type signature of `send`, any serializable data structure can be sent as a message to any process. Whether or not a particular message will be accepted (i.e. dequeued and acted upon) by the recipient is not determined until runtime. An alternative to sending messages by process identifier is to use *typed channels*, which use Haskell’s strong typing to make static guarantees that the sent message is of the right type.

Each distributed channel consists of two ends, which we call the *send port* and *receive port*. Messages are inserted via the send port, and extracted in FIFO order from the receive port. Unlike process identifiers, channels are parameterized by type and may contain only values of that type.

The central functions of the channel API are:

```
newChannel :: (Serializable a) ⇒ ProcessM (SendPort a, ReceivePort a)
sendChannel :: (Serializable a) ⇒ SendPort a → a → ProcessM ()
receiveChannel :: (Serializable a) ⇒ ReceivePort a → ProcessM a
```

A critical point is that although `SendPort` can be serialized and copied to other nodes, allowing the channel to accept data from multiple sources, the `ReceivePort` cannot be moved or copied from the node on which it was created. Fixing the position of the receiver greatly simplifies message routing, thereby ensuring that the programmer is able to control the cost model of network communication. This restriction is enforced by making `SendPort`, but not `ReceivePort`, an instance of `Serializable`.

We can now reformulate our ping example to use typed channels. The process must be given two ports: a receive port on which to receive pongs, and a send port on which to emit pings. Each `Ping` and `Pong` message now contains the send port on which its recipient should respond; thus the `Ping` message contains the send port of a channel of pongs, and the `Pong` message contains the send port of a channel of pings.

```
ping2 :: SendPort Ping → ReceivePort Pong → ProcessM ()
ping2 pingout pongin =
  do { Pong partner ← receiveChannel pongin
      ; sendChannel partner (Ping pongin)
      ; ping2 pingout pingin }
```

There is an analogy between the `expect` function and its channel-based counterpart, `receiveChannel`: both receive a message of a particular type. The channel-based coun-



terpart of `receiveWait` is the `mergePorts` family of functions, which let us receive a message from one of several channels.

```
mergePortsBiased :: Serializable a => [ReceivePort a] -> ProcessM (ReceivePort a)
mergePortsRR    :: Serializable a => [ReceivePort a] -> ProcessM (ReceivePort a)
```

Given a list of `ReceivePorts` of the same message type, these functions will return a new `ReceivePort` that, when read with `receiveChannel`, will provide a message taken from one of the input `ReceivePorts`. You can visualize that the `mergePorts` functions take several “feeder” ports and squeeze them together into a “merged” port. Even after producing the merged `ReceivePort`, the original ports may continue to be used independently. Messages read from the merged port are extracted from the queue of the feeder port, and so it is impossible to receive the same message twice.

The functions `mergePortsBiased` and `mergePortsRR` differ in the order that the input ports are queried, which is significant in the case that more than one port has a message waiting. Each subsequent read from the port created by `mergePortsBiased` will query the feeder ports in the order that they were provided to `mergePortsBiased`—so reading is “biased” towards the first feeder port, possibly leading to starvation of later ports. When that is not desirable, use `mergePortsRR` instead, which cycles through its feeder ports in “round-robin” order, ensuring that, given enough reads, every feeder port will eventually have a chance to contribute a message from its queue.

## 2.2 Processes

While we have already discussed how to send messages between processes, we have so far neglected to mention where these processes come from. To start a new process in a distributed system, we need a way of specifying where it will run. The question of *where* is answered with our framework’s unit of location, the node. We also need a way to say *what* will be run, that is, a way to identify the code to execute. The “what” in this case is a *closure*, a data structure that identifies a function and contains its environment (that is, its free variables). Therefore, Cloud Haskell includes `spawn` for starting new processes:

```
spawn :: NodeId -> Closure (ProcessM ()) -> ProcessM ProcessId
```

The `spawn` function takes a location (given by a `NodeId`) and closure identifying a `ProcessM`-monadic function returning unit. The `spawn` function itself returns the process identifier of the newly-created process.

The `spawn` function’s closure argument is parameterized by the return type of the underlying function; when the closure is invoked, a value of that type will be the result. In this case, the closure refers to a function returning `ProcessM ()`. Note that the parameters of the function to be called are not part of type signature; their values are encoded within the `Closure` data structure.

But where do these mysterious Closures come from? Each remotely-callable function has a corresponding *closure generator* function, named with a `_closure` suffix; the closure generator returns a closure suitable for use with `spawn`. The closure generators themselves are created automatically by a compile-time call to the `remotable` function (built with the Template Haskell metaprogramming facility [23]). Specifically, given a top-level user-defined non-polymorphic function  $f :: A \rightarrow B \rightarrow C$  (where  $A$  and  $B$  are serializable, and where  $C$  is not an arrow type), `remotable` will emit the closure generator  $f\_closure :: A \rightarrow B \rightarrow \text{Closure } C$ . To provide a concrete example, suppose that the programmer has written the following function:

```
processFile :: String → Int → [FilePath] → ProcessM ()
processFile s n f =...
```

To call `processFile` on a remote node, it suffices to write:

```
$( remotable ['processFile] )

dolt = spawn someNode (processFile__closure "hello" 3 ["pelda.txt"])
```

The `$( )` brackets above identify a Template Haskell splice: `remotable` takes a list of user-defined function names and in response it constructs their closure generators. In this case, it emits the definition of `processFile__closure`. Next, we call `processFile__closure` with the same arguments we would use as if we were calling `processFile` directly. The closure generator serializes its arguments, encodes a function representation, and returns a closure of type `Closure (ProcessM ())` (based on `processFile`'s original return type), which we then give to `spawn`. Finally, `spawn` will invoke `processFile` on the node identified by `someNode`. Thus, we can conveniently invoke functions through closures using syntax highly reminiscent of a straightforward function call.

We have shown how to use `spawn` to start a process remotely, but what if you want to perform a calculation remotely and then get its result? For that, we offer `callRemote`, which, like `spawn`, takes a closure. Unlike `spawn`, `callRemote` will block until the called function has completed, and then retrieve and return its result. In that way, the return value of `callRemote` depends on the type of the closure:

```
callRemote :: (Serializable a) ⇒ NodeId → Closure (ProcessM a) → ProcessM a
```

There are some restrictions on functions that can be called remotely. Clearly, all of the remotely-callable function's parameters must be `Serializable`. Its return type should be either pure, or be in the `IO`, `TaskM`, or `ProcessM` monad. Only top-level functions are remotely callable. A more serious restriction is that remote functions may not be polymorphic, because the type of the function must be known statically in order for the framework to deserialize its arguments.

We discuss the mechanism of remote function invocation in greater detail in Section 4.4.

## 2.3 Fault monitoring

A key feature of a distributed system is *fault tolerance*, the ability to continue operating normally in the face of the failure of some component, especially hardware. Applications designed for execution on a single system do not need to worry about fault tolerance: the hardware either works, or it doesn't. But in a large enough distributed deployment, the failure of a hardware component will be a routine occurrence.

Fault tolerance has two parts: *fault monitoring*, in which the failure of a component is discovered; and *fault recovery*, in which appropriate corrective action is taken. These are separable problems: detecting hardware failure is a concept general to most applications, whereas recovering from failure is often very application-specific. For example, some programs may have to “undo” partially applied actions or otherwise clean up after failure.

Cloud Haskell aims to accommodate most kinds of applications by providing mechanisms for both fault monitoring and fault recovery. In this section, we concern ourselves only with fault monitoring. Cloud Haskell's fault recovery is integrated with its task layer and is discussed in Section 2.4.1.

Erlang provides the immediate inspiration for Cloud Haskell's fault monitoring interface, and we closely follow its model. The framework lets one process monitor another. If the monitored process fails, the monitoring process will be notified, whereupon it may take application-specific action. Monitoring is thus a (non-symmetric) binary relation between processes. This relation can also be considered to express dependency: that is, the monitoring process depends on the monitored process.

We distinguish two ways that a process may fail. Besides the motivating case of hardware failure, a process may fail by software error. In Cloud Haskell, a process that lets a thrown exception propagate uncaught will be considered to have failed. This lets the programmer apply the same process monitoring mechanism that deals with hardware failures to common software errors, such as divide-by-zero.

Monitoring of hardware faults takes place at the node level. The node of the monitoring process sends regular pings to the nodes of the monitored process; if the monitored node fails to respond within a time limit, then the node, as well as all process on it, is judged to have failed, and any process monitoring a process on the failed node will be duly notified.

Notification of a failure may take one of two forms: either the monitoring process will be sent a message, or it will be sent an asynchronous exception. The form of notification is specified when the process requests monitoring. If the monitoring process is sent an asynchronous exception, it can handle the event by catching the exception using Haskell's usual exception primitives. If the monitor is notified by message, it can retrieve the message using the the message handling functions described in Section 2.1.1.

Cloud Haskell provides functions for establishing and dissolving monitoring:

```

data MonitorAction = MaMonitor | MaLink | MaLinkError
data SignalReason = SrNormal | SrException String | SrNoPing | SrInvalid
data ProcessMonitorException = ProcessMonitorException ProcessId SignalReason

```

```

monitorProcess :: ProcessId → ProcessId → MonitorAction → ProcessM ()
unmonitorProcess :: ProcessId → ProcessId → MonitorAction → ProcessM ()

```

The call `monitorProcess monitor monitee action` will notify the process monitor when the process `monitee` ends. The `action` parameter describes how the monitor will be notified:

- `MaMonitor` – the monitoring process will be sent a message of type `ProcessMonitorException` when the monitored process ends
- `MaLink` – the monitoring process will be sent as asynchronous exception of type `ProcessMonitorException` when the monitored process ends
- `MaLinkError` – the monitoring process will be sent as asynchronous exception of type `ProcessMonitorException` when the monitored process ends abnormally; that is, because of an uncaught exception or because its node is not responding to pings

The reason for the failure is given by the `SignalReason` field in `ProcessMonitorException`.

The `unmonitorProcess` function destroys the relation between two processes; after a call to `unmonitorProcess`, the monitoring process will not be notified when the monitored process ends. Multiple calls to `monitorProcess` will not result in repeated notifications, but a corresponding number of calls to `unmonitorProcess` will be required to definitively dissolve monitoring.

Cloud Haskell also provides higher-level convenience functions related to process monitoring:

```

linkProcess :: ProcessId → ProcessM ()
spawnLink :: NodeId → Closure (ProcessM ()) → ProcessM ProcessId

withMonitor :: ProcessId → ProcessM a → ProcessM a
matchProcessDown :: ProcessId → ProcessM q → MatchM q ()

```

Here, `linkProcess` is an analogue to Erlang’s `link` function: it establishes bidirectional monitoring of abnormal termination between the current process and the indicated process. `linkProcess` can be defined in terms of `monitorProcess`. `spawnLink` is a variant of `spawn` that, in addition to creating a new process, begins monitoring it, like Erlang’s `spawn_link`.

The `withMonitor` function can be used to introduce a block within which the given process is monitored. It accomplishes this by first calling `monitorProcess`, then invoking the user-provided block, and finally calling `unmonitorProcess`. At the end of the transaction, the

monitoring relationship between the two processes is the same as it was initially. Within the provided block, the programmer should call `receiveWait` with `matchProcessDown`, which is a variant of `match` (described in Section 2.1.2) specialized for receiving `ProcessMonitorException` messages.

## 2.4 Tasks

The interface described thus far, concerned with direct manipulation of processes and messages and accessible via the `ProcessM` monad, is called the *process layer*. The abstractions of the process layer are general but rudimentary. We now present a higher-level Cloud Haskell interface which provides additional functionality, at the price of flexibility. This interface, the *task layer*, is implemented in terms of the process layer.

The advantage of the higher-level task interface are:

- An automatic mechanism for fault recovery. Whereas the process layer puts the burden of recovery from a hardware failure on the application developer, the task layer will automatically restart failed calculations, in a way transparent to the user.
- Data-centric calculation model. While the process is an abstraction for location-independent calculations, central to the task layer is the data that is produced from such calculations, which can then be manipulated by the programmer regardless of where the calculation was performed and even if it has finished yet.

While the process layer draws its inspiration primarily from Erlang, the design of the task layer is influenced by the Skywriting/CIEL [16] [17] project. A key difference is that although the CIEL system distinguishes a “coordination” language from a “calculation” language, our task layer allows the programmer to both describe data flow and perform calculations in a single language, i.e. Haskell.

### 2.4.1 Interface

Like the process layer, the task layer is a domain-specific language embedded in Haskell. In fact, it is embedded in the DSL provided by the process layer. The central abstraction is the *promise* (in Skywriting, known as a *future*), which represents the result of a calculation that may or may not have yet completed. A promise works like a read-only reference: it can be cheaply passed about, and an attempt to dereference the promise will block until the value is available.

A calculation that produces the value of a promise is a *task*, which execute in the `TaskM` monad. `TaskM` builds on the `ProcessM` monad, but also restricts the range of allowable

```

avg :: [Integer] → TaskM Integer
avg xs = return (sum xs `div` fromIntegral (length xs))

diff :: Promise Integer → Promise Integer → TaskM Integer
diff pa pb =
  do { a ← readPromise pa
      ; b ← readPromise pb
      ; return (a - b) }

$( remotable ['avg', 'diff'] )

initialProcess "MASTER" =
  do { res ← runTask $
      do { p1 ← newPromise (avg__closure [0..50])
          ; p2 ← newPromise (avg__closure [50..100])
          ; p3 ← newPromise (diff__closure p1 p2)
          ; readPromise p3 }
      ; say ("Result: " ++ show res) }
initialProcess _ = receiveWait []

```

Figure 2.2: An example program using the task layer.

actions to those for manipulating promises. In particular, the programmer cannot escape the monad to execute arbitrary IO actions. These restrictions are necessary to ensure that each task can be safely restarted in the event of failure; this would not be the case if tasks could write to disk files, set MVars, or send messages. The guarantee of idempotency is possible thanks to Haskell’s purity and monads; in a lesser language, such guarantees would not be statically enforceable.

The `newPromise` function creates a new promise and starts a task that will calculate its value:

```
newPromise :: (Serializable a) ⇒ Closure (TaskM a) → TaskM (Promise a)
```

It accepts a closure, which identifies the function that will eventually provide a value for the returned promise. The calculation given in the closure is started concurrently, and so `newPromise` returns immediately. Promises are write-once: a promise acquires a value only once, when its associated task returns. The promise returned from a call to `newPromise` can be dereferenced with `readPromise`:

```
readPromise :: (Serializable a) ⇒ Promise a → TaskM a
```

If the value wrapped by the promise is available on the current node, it will be returned immediately. If the value is available on another node, it will be copied to the current node and returned. If the task generating the promise has not completed yet, then `readPromise`

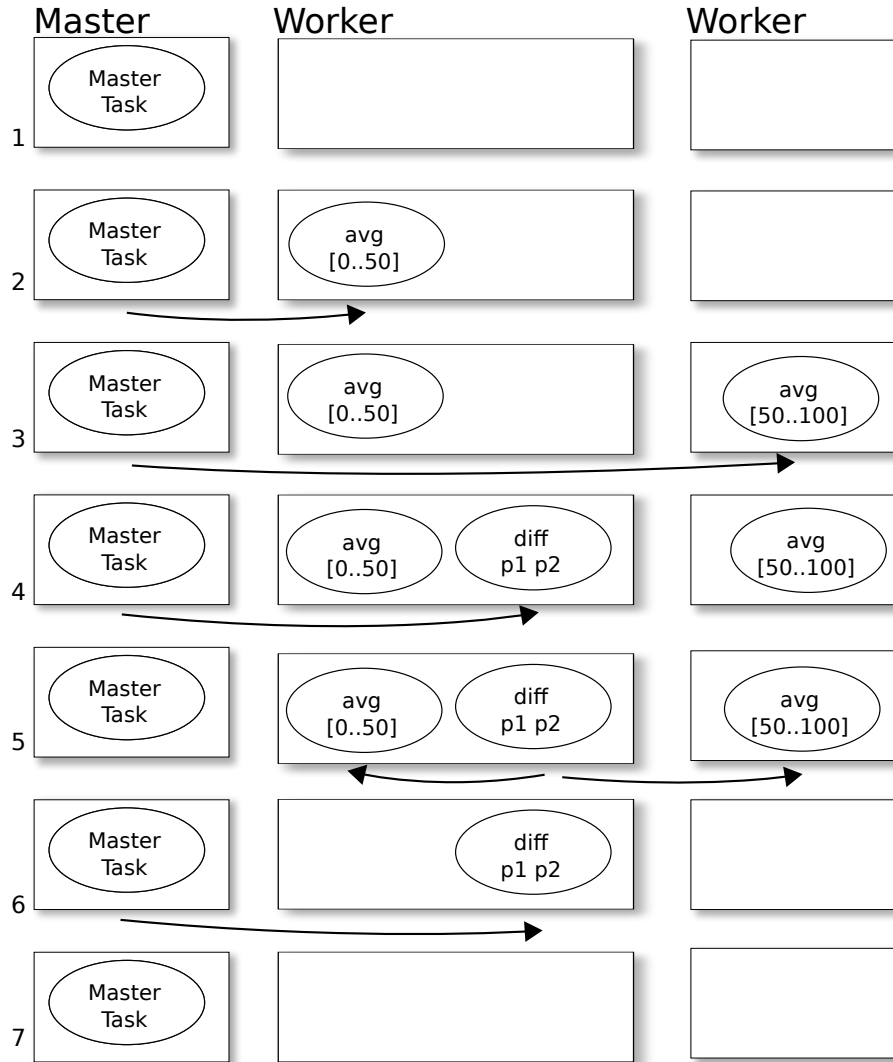


Figure 2.3: The propagation of promises in the program from Figure 2.2. Initially, worker nodes are idle (1) until the master task assigns `avg` tasks to them (2) (3), yielding the promises `p1` and `p2`. Those promises are passed to a call to `diff` (4), yielding promise `p3`. The `diff` task calls `readPromise` to query the values of `p1` and `p2`, which blocks until their values are finalized (5). Finally, the master task evaluates promise `p3` (6), yielding a final result. The worker nodes are again quiescent (7).

blocks until it has and returns its result. Because the value is not guaranteed to be available until it is read, we can consider promises to ask like a form of explicit lazy evaluation.

Although the allocation of tasks to nodes is decided by the framework, making the shape of the network transparent to the programmer, the framework accepts hints about where a given a task should run. This is useful for cases when the programmer knows that two tasks will exchange a large amount of data, and can therefore benefit in performance by reducing the network traffic between them. For such cases, we provide variants of `newPromise` that accept locality suggestions:

```
newPromiseHere :: (Serializable a) ⇒ Closure (TaskM a) → TaskM (Promise a)
newPromiseNear :: (Serializable a, Serializable b) ⇒ Promise b →
```

```

Closure (TaskM a) → TaskM (Promise a)
newPromiseAtRole :: (Serializable a) ⇒ String → Closure (TaskM a) →
TaskM (Promise a)

```

These functions will try to run the task on, respectively, the current node; the same node as another given promise; or on some node having the given role. In order to preserve the property of determinism in tasks, the user application is not notified whether its locality hint is honored or not; the actual allocation of tasks to nodes remains transparent.

In the example program in Figure 2.2, which calculates the difference between the averages of two lists of integers, we see how the task layer shares with the process layer a mechanism for creating and calling closures. We also introduce the `runTask` function, which introduces a `TaskM` context from within the `ProcessM` monad. In Figure 2.3, we trace the execution of this program. Like all programs built with Cloud Haskell’s task layer, it is implicitly fault-tolerant. A fault in a worker node will be detected by the framework and any incomplete calculations will be restarted, without any explicit direction from the programmer. This is safe because of the idempotent guarantee enforced by the `TaskM` monad: tasks are only allowed to perform pure calculations and manipulate promises. In other words, each task is deterministic and may produce only one result, so restarting it cannot leave the calculation in an indeterminate state. For now, this fault tolerance, however, does not extend to the master node.

The task layer’s fault recovery relies on the process layer’s fault monitoring mechanism, described in Section 2.3. The implementation of the task layer is described in Section 4.3.

## 2.4.2 MapReduce

MapReduce [5] is a well-known framework for distributed computations on large data sets. It has served as a model for open-source frameworks, such as Hadoop. Cloud Haskell’s task layer is sufficiently powerful that a MapReduce-like algorithm can be concisely implemented in just a few lines of code. Here we give the complete implementation:

```

mapReduce :: (Serializable i, Serializable k, Serializable m, Serializable r) ⇒
  MapReduce ri i k m r → ri → TaskM [r]
mapReduce mr inputs =
  let chunks = (mtChunkify mr) inputs
  in do { pmapResult ← mapM (λchunk → newPromise ((mtMapper mr) chunk) )
        chunks
        ; mapResult ← mapM readPromise pmapResult
        ; let shuffled = (mtShuffle mr) (concat mapResult)
        ; pres ← mapM (λmid2 → newPromise ((mtReducer mr) mid2)) shuffled
        ; mapM readPromise pres }

```

`mapReduce` takes four user-specified functions as arguments:



- `mtChunkify` – breaks up raw input into chunks, where each chunk is assigned to a mapper
- `mtMapper` – convert a chunk of input data into an intermediary result
- `mtShuffle` – assign intermediary results to appropriate reducers, usually using a key encoded in the intermediary result
- `mtReducer` – convert sets of intermediary results to final results

For example, imagine a distributed application for counting occurrences of words in a text file. Such an application could use the default versions of `mtShuffle` and `mtChunkify`, and would provide a mapper and reducer similar to the following:

```
mrMapper :: [Line] → TaskM [(Word, Int)]
mrMapper lines =
    return (concatMap (\line → map (\w → (w,1)) (words line)) lines)

mrReducer :: (Word,[Int]) → TaskM (Word,Int)
mrReducer (w,p) =
    return (w,sum p)
```

The mapper simply breaks each group of lines into words, marking each word as having a single occurrence. The words are then used as keys, so that all intermediary results for a given word are sent to the same reducer. The reducer then need only sum the counts of each word. The final result will be the total number of occurrences in the file.

This implementation of MapReduce is also used by our *k*-means demo application, described in Section 5.1.



# Chapter 3

## A practical tour

We now construct a simple but nontrivial distributed application using Cloud Haskell’s process layer API. We will also see how to deploy this application in a real-world environment.

The program, based on a similar example packaged with the Hadoop framework, will calculate and display an estimate of the value of  $\pi$ , and is implemented in Cloud Haskell’s process layer. Internally, the program plots a sequence of evenly-distributed points on the unit square, and counts how many of those fall within the unit circle; the ratio of points within the circle to the total number of circles will give an estimate of  $\pi$ . It should be noted that while this is an instructive example, it is by no means the most efficient way to estimate  $\pi$ . This simple example does not use any of Cloud Haskell’s features for fault tolerance.

Our application can run on an arbitrary number of nodes. One of these will be designated the master node, and will coordinate the activities of the other nodes, which are the workers. The program uses a Monte Carlo-like method to plot the points. To distribute the calculation, the master node assigns each worker node a non-overlapping range from the Halton sequence, a quasi-random number sequence. The workers plot and count the points, and then transmit their findings to the master, which accumulates the total sum from all workers. When each worker has reported, the master generates an estimate of  $\pi$  from the returned values and displays it.

### 3.1 Code

The complete source code for this example can be found in the Cloud Haskell distribution as `Pi6.hs`. Here we discuss excerpts of the code.

```
{-# LANGUAGE TemplateHaskell #-}  
module Main where
```

```
import Remote
```

We enable the GHC-specific Template Haskell extension, which are needed for automatic closure generation. The `Remote` module contains Cloud Haskell.

```
haltonPairs :: Int → [(Number,Number)]
...
```

We elide the implementation of the `haltonPairs` function. It suffices to know that `haltonPairs n` will generate an infinite list of coordinates in the range  $(0,1) \times (0,1)$ , starting with the coordinate pair composed of the  $n$ th element of the second and third Halton sequences.

```
countPairs :: Int → Int → (Int,Int)
countPairs offset count =
  let range = take count (haltonPairs offset)
      numout = length (filter outCircle range)
  in (count-numout,numout)
where
  outCircle (x,y) =
    let fx=x-0.5
        fy=y-0.5
    in fx*fx + fy*fy > 0.25
```

Given a finite range in the Halton sequence, `countPairs` uses `haltonPairs` to generate the corresponding coordinates and counts how many of them fall within the unit circle. It returns a tuple of the number of points that fell within the circle and those that did not.

```
worker :: Int → Int → ProcessId → ProcessM ()
worker count offset master =
  let (numin,numout) = countPairs offset count
  in do send master (numin,numout)
      logS "PI" LsInformation ("Finished mapper from offset "++show offset)

$( remotable ['worker] )
```

The `worker` function is invoked by the master node on each of the worker nodes. Its parameters `count` and `offset` identify a range in the Halton sequence. After calling `countPairs`, it sends the result back to the master node and emits a log message.

```
initialProcess :: String → ProcessM ()

initialProcess "WORKER" =
```

```

receiveWait []

initialProcess "MASTER" =
  do { peers ← getPeers
      ; mypid ← getSelfPid
      ; let { workers = findPeerByRole peers "WORKER"
          ; interval = 1000000
          ; numberedworkers = (zip [0,interval..] workers) }
      ; mapM_ (λ (offset,nid) → spawn nid (worker__closure (interval-1) offset
          mypid)) numberedworkers
      ; (x,y) ← receiveLoop (0,0) (length workers)
      ; let est = estimatePi (fromIntegral x) (fromIntegral y)
          in say ("Done: " ++ longdiv (fst est) (snd est) 20) }
  where
    estimatePi ni no | ni + no == 0 = (0,0)
                    | otherwise = (4 * ni , ni+no)
    receiveLoop a 0 = return a
    receiveLoop (numIn,numOut) n =
      let
        resultMatch = match (λ (x,y) → return (x::Int,y::Int))
      in do { (newin,newout) ← receiveWait [resultMatch]
          ; let { x = numIn + newin
              ; y = numOut + newout }
          ; receiveLoop (x,y) (n-1) }

initialProcess _ = error "Role must be WORKER or MASTER"

```

The `initialProcess` function is the shared entry point of all nodes. It is given as a parameter the *role* assigned to that node. In this application, as discussed above, we distinguish two roles: that of master, and of worker.

Worker nodes are initially passive, and simply await instruction from the master node.

The master node calls the `getPeers` function to obtain a `PeerInfo` structure, which it queries for worker nodes; peer discovery is covered in Section 3.2 and Section 4.2.4. We then iterate over worker nodes, invoking `worker__closure` on each. Then the master process retrieves the workers' results with `receiveWait`. Once all workers' results have been harvested, the master calculates an estimate of  $\pi$  in `estimatePi` and displays the result with `say`.

Because the master process uses all available worker nodes, and each worker is assigned a fixed number of points to analyze, adding more nodes to the application will increase the accuracy of the result.

```

main :: IO ()
main = remotelInit (Just "config") [Main._remoteCallMetaData] initialProcess

```

Finally, `main`, the entry point of the program, needs to initialize the Cloud Haskell framework with a call to Cloud Haskell’s `remotelnit` function. It takes three parameters: an optional configuration file; a list of all the function lookup tables generated by `remotable`; and the function for starting the first process.

## 3.2 Deployment

Now, we show how to run this program on a network. The behavior of the framework is controlled by several key/value pairs called *configuration options*. A complete accounting of configuration options is given in Appendix A.

Configuration options are specified on a per-node basis, usually in a configuration file named `config`. Each node can have its own configuration file. A Cloud Haskell application may be deployed onto a local area network, as we describe below, but it can also be hosted on a third-party cloud environment, such as Amazon EC2.

Each `NodeId` contains routing information sufficient for another node to send messages to processes on that node. While the structure of a `NodeId` is opaque to the application itself, it is germane to a discussion of network configuration. A `NodeId` has the form `nid://velikan:44632/`, where `velikan` is the hostname of the computer on which the node is running and `44632` is TCP port on which that node is listening for messages. `ProcessIds` have a similar structure: a process running on the node given above might be identified as `pid://velikan:44632/23/`. Here, `23` is the locally unique process number. The administrator of a Cloud Haskell deployment must ensure that the given address and port are reachable and open.

In this example, we assume that the network consists of three hosts, named `host1`, `host2`, and `host3`. The application’s executable must be available on each host.

Although we only have three hosts, we are not limited to three nodes, because a single host can support an arbitrary number of nodes. In this deployment, we will run the master node and one worker node on `host1`; two worker nodes on `host2`; and one worker node on `host3`.

Next, we need to create a configuration file for each node. The only configuration option that is strictly necessary is `cfgRole`. As a best practice, though, we assign values to other configuration values, as well. Below, we give the recommended configuration files for the five nodes in this example.

```
# config file for master node on host1
cfgRole MASTER
cfgHostName host1
cfgKnownHosts host1 host2 host3
```

```
# config file for worker node on host1
cfgRole WORKER
cfgHostName host1
cfgKnownHosts host1 host2 host3

# config file for both worker nodes on host2
cfgRole WORKER
cfgHostName host2
cfgKnownHosts host1 host2 host3

# config file for worker node on host3
cfgRole WORKER
cfgHostName host3
cfgKnownHosts host1 host2 host3
```

The `cfgKnownHosts` option gives a list of hosts where to search for nodes, and all nodes found on any of those hosts will be available to the application via `getPeers`.

What if we do not want a node to be used? For example, we may have two distinct Cloud Haskell applications running on the same network, and maybe even sharing hosts. We need to prevent the nodes designated for one application from being used by the other. To clearly distinguish which nodes “belong” to which application, use the `cfgNetworkMagic` option. Nodes with differing values of this option will refuse to communicate with each other.

When a node starts, it begins listening on a port for incoming messages. The port that a node listens on can be set manually with `cfgListenPort`, but if left unspecified, the operating system will assign an arbitrary available port. Each node publicizes its availability, including its port, on a *node registration server*, which is started automatically. In addition, you can explicitly start a node registration server with the `RegServ` utility included in the distribution. For troubleshooting network communication issues, the distribution also includes a diagnostic program `Diag`, which provides information about what nodes are currently visible.

Having created configuration files, we start the application. First, run the `Pi6` program on the worker nodes, as they need to be available when the master starts. Then, run the master node. After a short time, it should emit text similar to this:

```
2011-05-28 17:24:46.158548 BST 0 pid://velikan:44632/7/      SAY Done:
31415071415071415071
```

This output is formatted by Cloud Haskell’s logging facility, described in Section 4.2.3. From left to right, we see: the date and time that the message was produced; the priority of the message; the identifier of the process that generated the message; the application

component responsible for the message; and the text of the message, which here contains a not particularly accurate estimate of  $\pi$ .

After producing this output, the master node terminates. The worker nodes do not terminate, because they are still waiting for requests. In a real application, they could be configured to end after processing has completed. In this case, they need to be terminated manually.

A similar procedure can be used for deploying a Cloud Haskell application on Amazon EC2. One important difference is that the hostnames of the cloud servers are assigned when you request the virtual machine instances. Fortunately, there are scriptable interfaces to Amazon's service and it is easy to generate a set Cloud Haskell configuration files based on the dynamically-assigned host names. The `awsgo` script included in the Cloud Haskell distribution provides a demonstration of this technique.

For testing purposes, it may be useful to run a distributed application on a single computer, while keeping the semantics of multiple nodes. This program, like all Cloud Haskell applications, can run as an arbitrary number of nodes on a single machine with only minor adjustments to the configuration files.



# Chapter 4

## Implementation

In this chapter we discuss some interesting technical challenges that we faced in implementing the framework. While this discussion is strategic, the finer points assume some familiarity with Haskell’s facilities for concurrent programming and software transactional memory [9].

### 4.1 Message handling

The central feature of Cloud Haskell’s process layer, described in Chapter 2, is the `ProcessM` monad, and the central feature of the `ProcessM` monad is the ability to send and receive messages. Here we describe our implementation of message transmission.

A process identifier contains a hostname, TCP port, and a unique local process number. When sending a message to a process, the sending process connects to the given port, identifies the target process by number, and sends the serialized message payload. At this point the message has been *received* by the remote node. Once received, the message is put into a message queue specific to that process, and so the message is *delivered*. The message will stay in the queue until it is *extracted* with a matching call to `receiveWait` or `expect`. The typed channels described in Section 2.1.3 share the underlying mechanism.

You might expect that we could skip the serialization step when sending a message to a process on the same node. However, consider the command `send somePid [undefined]`. If we serialize this message, the use of `undefined` will cause an exception on the sender’s side. If we skip serialization, lazy evaluation will mean that the exception will occur in the receiving process, which may not be prepared for it. Thus, to preserve the semantics of messaging, we use serialization to force full evaluation of all messages in the sending process.

The message queue itself is implemented as a `TChan`, from Haskell’s software transactional memory (STM) library. This library lets us compose atomic operations on the queue.

In particular, each call to `match` in a `receiveWait` block atomically iterates through the delivered messages and removes the first matching one. The ability to compose atomic operations is also necessary when working with channels, especially in the `mergePorts` family of functions, which needs to perform a blocking read operation on several feeder channels simultaneously, from which only one message should be extracted. If a message is received by more than one channel at the same time, it is precisely the transactional feature of `TChan` that lets `mergePorts` “put back” one of the messages, preventing message loss that could occur if more than one message were removed.

## 4.2 System processes

In addition to managing user-created process, the Cloud Haskell framework maintains several additional processes for various services on each node. These processes are started automatically by the framework and their presence is invisible to user applications. Nevertheless, they underlie much of the functionality of the framework. The most important of these system processes are the Spawner (Section 4.2.1), the ProcessMonitor (Section 4.2.2), the Logger (Section 4.2.3), and the Discovery process (Section 4.2.4).

### 4.2.1 Spawner

Cloud Haskell’s processes are built on Haskell’s lightweight `forkIO` threads. Those threads, though, cannot be directly started across a network. To be able to start a process on a remote node, we need to send a message containing the appropriate closure. Such messages are sent to the Spawner process. The user application never needs to refer to the Spawner process directly, since this message is sent by the `spawn` function, which will then block until it receives the process identifier of the newly-created process from the remote node’s Spawner. An illustration of the process is given in Figure 4.1.

### 4.2.2 Process monitor

As discussed in Section 2.3, when a process calls `monitorProcess`, a relation is created such that one process will be notified when another terminates. The state of these relations is administered by Cloud Haskell’s administrative ProcessMonitor process. Like other system processes, user applications never interact with process directly, but only through primitives exposed in the Cloud Haskell API.

In the simplest case, both the monitoring process and the monitored process are on the same node. The call to `monitorProcess` will send a message to the ProcessMonitor, causing it to update its state. When the monitored process ends, it will also send a message to the ProcessMonitor, which will notify each of the monitoring processes in their preferred

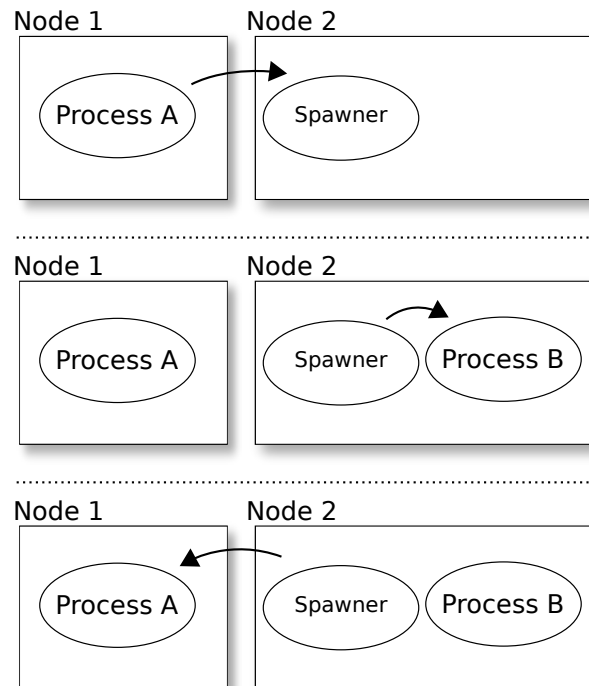


Figure 4.1: The Spawner process in action. The `spawn` function invoked on Node 1 sends a message to the spawner process on Node 2. The Spawner process creates the new process. Finally, the Spawner process sends the PID of the new process back to the parent process. That PID is returned by the user application.

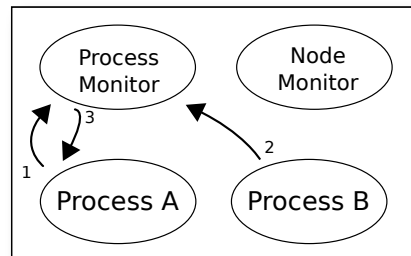


Figure 4.2: Process monitoring on one node. Process A initiates monitoring of Process B by sending a monitor request (1) to the local ProcessMonitor administrative process. From that point, Process A is a registered monitor of Process B. When Process B terminates, ProcessMonitor will receive a message (2), and will subsequently notify Process A, either by message or by asynchronous exception (3).

manner. Because all processes are running on the same node, we need to consider only “soft” termination; that is, cases when the process ends normally or with an uncaught exception, rather than by catastrophic hardware failure. The flow of messages when monitoring a local process is shown in Figure 4.2.

When the monitored process is on a different node from the monitoring process, though, we also need to take into account the possibility of a “hard” failure; that is, caused by a hardware failure. To that end, the ProcessMonitor interacts with another system process, the NodeMonitor. When monitoring of a remote process is requested, the ProcessMonitor sends a message to NodeMonitor requesting that it send regular pings to that node. If the NodeMonitor finds that its pings are not being answered, it will send a message back

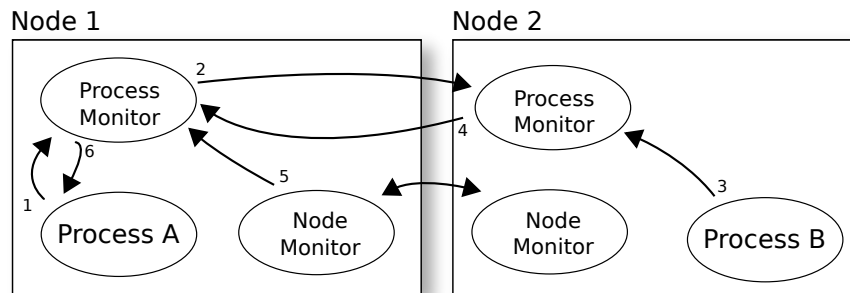


Figure 4.3: Process monitoring between nodes. As in Figure 4.2, Process A initiates monitoring of Process B by sending a monitor request (1) to the ProcessMonitor on Node 1. Because Process B is on a different node, Node 1’s Process Monitor requests monitoring from Node 2’s ProcessMonitor (2). When Process B terminates, Node 2’s ProcessMonitor will receive a message (3), and will subsequently notify Node 1’s ProcessMonitor (4), which will finally notify Process A, either by message or by asynchronous exception (6). Alternatively, if Node 2 fails completely, Node 1’s NodeMonitor administrative process will fail to receive a ping from Node 2’s NodeMonitor, in which case it will send a message to Node 1’s ProcessMonitor (5). The ProcessMonitor will then notify all local processes that were monitoring any process on the downed node (6).

to its ProcessMonitor, which will then notify all processes monitoring any process on the downed node. In addition, the case of a “soft” process termination is somewhat more complicated than with a single node: the ProcessMonitors on both nodes must coordinate their activity, so that the ProcessMonitor on the monitored process’s node waits for its termination, and then sends a message to the ProcessMonitor on the monitoring process’s node, which then passes on the notification to its user processes. This flow of messages is shown in Figure 4.3. This complexity is hidden from the user: whether monitoring a local or remote process, and whether this process ends by a “hard” or “soft” method, the user application uses the same primitives and is notified in a similar way. The only way a user application can distinguish these cases is by examining the `SignalReason` field of its `ProcessMonitorException` message. This opacity is important to ensure that process semantics are preserved regardless of the details of deployment.

The internal complexity of the ProcessMonitor is increased by the need to maintain the semantics of `monitorProcess`, which should not return until monitoring has been engaged. Otherwise, we could cause a race condition between a process’s termination and a request to monitor it, with the result that the monitoring process would never be notified and could deadlock. Therefore, `monitorProcess` must behave synchronously, but ProcessMonitor may not, since it needs to continue servicing requests from other processes.

### 4.2.3 Logger

To aid debugging and development, Cloud Haskell provides a configurable facility for logging. This lets programmers emit information messages to a system log, which can be filtered by importance and subsystem. By default, log messages are printed to `stdout`, but

the real value of this feature is in *log forwarding*: a node can be configured so that its log messages are sent to another node. In a distributed application consisting of dozens or hundreds of nodes, it would be prohibitive to check each node for error messages. The ability to forward log messages means that all messages can be conveniently displayed on a single computer. Log recording and forwarding is handled by the special Logger process.

```
type LogSphere = String
data LogLevel = LoSay | LoFatal | LoCritical | LolImportant | LoStandard |
               LolInformation | LoTrivial
logS :: LogSphere → LogLevel → String → ProcessM ()
say  :: String → ProcessM ()
```

The `logS` function emits a message to the Logger process, which then filters and forwards it according to the node’s current settings. The parameters to `logS` let the programmer indicate the importance of the message and the component that it applies to. By default, messages issued with importance below `LoStandard` will be suppressed.

The `say` function, like `logS`, emits a message to the system log. The difference is that `say` is intended for outputting messages to the user, rather than strictly technical debugging messages. Therefore, messages output by `say` are not subject to filtering by importance.

In applications built with Cloud Haskell’s task layer, all log messages are forwarded to the master node. In an application based on the process layer, the programmer must configure logging explicitly.

## 4.2.4 Discovery

Before a distributed application can communicate with a remote node, it must first know that it exists. From the programmer’s perspective, it suffices to call `getPeers`, which returns a structure containing information about all known nodes. But how does `getPeers` get this information? It uses two methods, and combines their results:

With *static peer discovery*, `getPeers` examines the nodes currently running on a fixed list of hosts, specified by `cfgKnownHosts`. It does this by sending a message to the *node registration server* of each host, with which each node must register when it starts, and unregister just before it ends. The number of nodes returned by this process is not limited, but only the specified hosts are searched. The list of hosts is static, and must be known when the application starts.

On the other hand, with *dynamic peer discovery*, `getPeers` is able find nodes running on hosts that are not mentioned in the application’s configuration. It accomplishes this by sending a UDP broadcast message to all hosts on the local network. Any Cloud Haskell nodes running on those hosts will respond by sending a message to the originating process. Thus, new hosts can be connected to the network during the life of an application, and

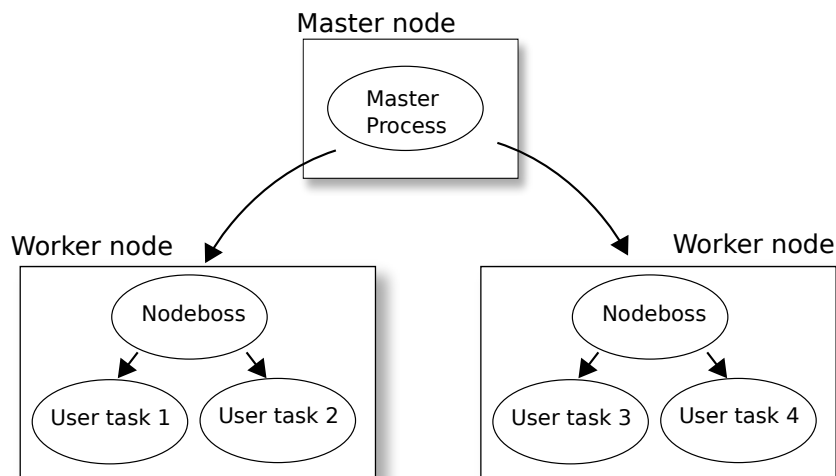


Figure 4.4: A schematic of the control flow in the task layer. The master process directs new promise creation to a worker node. Each worker node is controlled by a node boss process, which manages the creation of tasks and redemption of promises.

the application will be able to use them without having to restart. Because letting a distributed application communicate with nodes that are not explicitly on its “white list” can be a security problem, dynamic peer discovery can be disabled, as described in Appendix A. Responding to dynamic peer discovery requests is handled by the special Discovery process.

## 4.3 Tasks

It is easy to see that `newPromise`, described in Section 2.4, is based on `spawn`: it starts a concurrent calculation from a closure. However, the task layer also has to manage the values produced by such calculations, and ensure that they are accessible when another task retrieves them with `readPromise`.

In addition to user-level tasks, there are two kinds of administrative processes in the task layer: a single *master process*, which is responsible for allocating user tasks to nodes; and *node bosses*, which manage the promises available on a node. Each node has exactly one node boss. The relationship between the various administrative processes is shown in Figure 4.4. The master process is started by a call to `runTask`, which should be called only once per application. Node bosses are started by the master process when it discovers available worker nodes, and automatically terminate when they detect that their master process ends.

A call to `newPromise` sends a request containing a closure to the master process, which allocates a new unique `Promiseld`, selects an available node, and sends a message to that node’s node boss, which starts the task. At the same time, the master process stores the closure, so that it can be restarted if necessary. The master process returns a promise to the calling task. The implementation of a promise is:

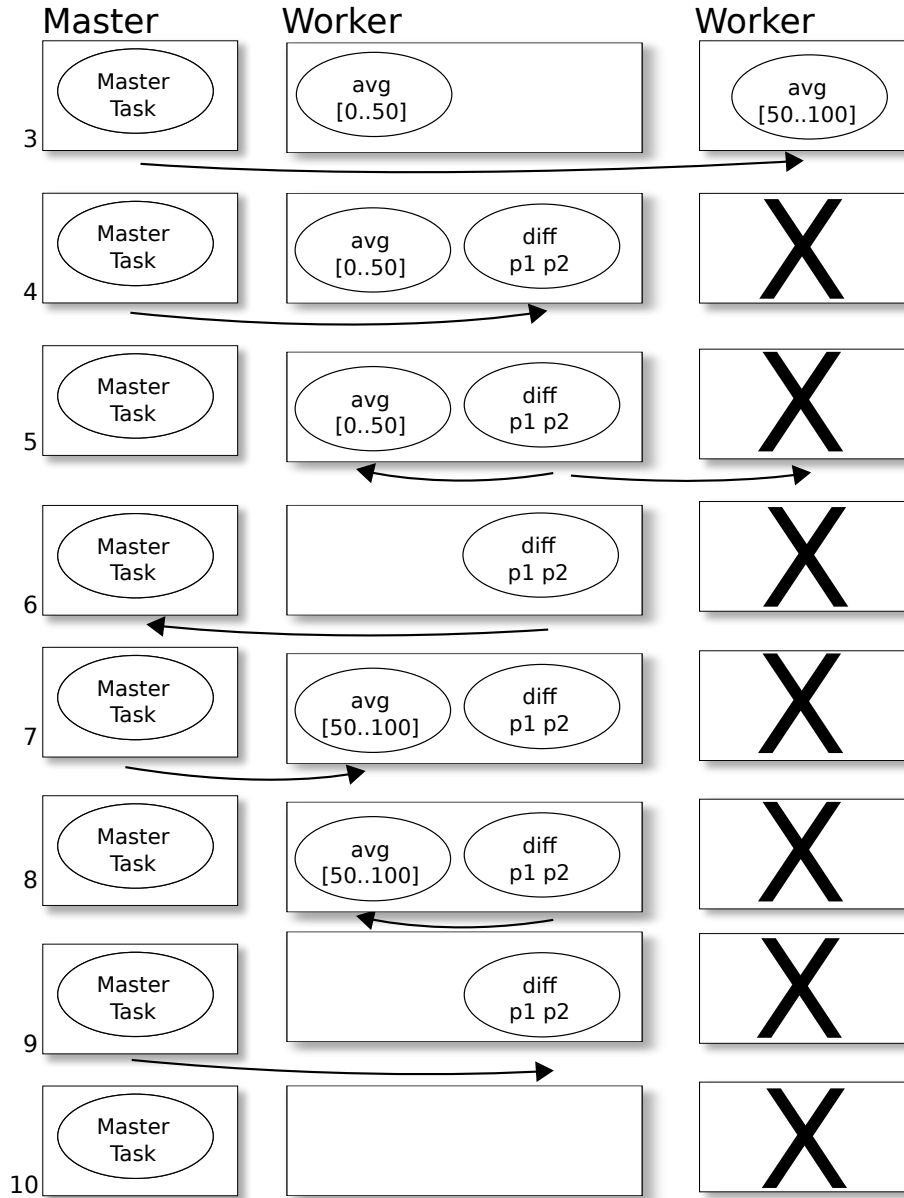


Figure 4.5: The propagation of promises in the presence of hardware failure. The program runs as in Figure 2.3 until the second worker node fails (4). When **diff** is unable to contact it (5), it complains to the master (6), who reissues the missing task on an available node (7). The **diff** task resumes where it left off, waiting on the result of the second **avg** task (8). Afterwards, the program concludes normally (9) (10).

```
data Promise a = Promise { psRedeemer :: ProcessId, psId :: PromiseId }
```

The **psRedeemer** field identifies the node boss responsible for starting the promise's task, and the **psId** field contains the unique promise identifier. When the task has completed, it will store its result in a local result cache, which is shared by all tasks running on that node. If a value in the promise cache is not read within a certain period of time, it will be flushed to disk in order to conserve memory.

A call to **readPromise** proceeds as follows: first, it checks the local result cache, where the value of the promise may already be stored, either from a previous call to **readPromise**, or

where it will be deposited if the task is running on the same node. Otherwise, `readPromise` attempts to contact the node boss identified in the promise itself, sending it the promise's `Promiseld`. The node boss will determine if the task has completed; if so, it will send its value from its own result cache, and otherwise it will wait until such a result is available. Upon receiving the value, `readPromise` returns it to the calling task.

What if the node boss named in the promise is not available? If `readPromise` cannot redeem a promise, it will assume that the corresponding node has crashed, and will complain to the master process. The master process, which has retained the closure originally used to start the task, will restart it, and send its new node boss to the requester. If multiple tasks complain about the same node boss, it will send them all to the same replacement, ensuring that a closure is not restarted more than necessary. The process is graphically represented in Figure 4.5.

## 4.4 Closures

In Section 2.2, we described the Cloud Haskell interface for creating and invoking closures. We now describe the implementation of this mechanism.

Conceptually, we can imagine a Cloud Haskell closure as this data structure:

```
data Closure a = Closure functionrep environmentstore
```

where the type `functionrep` identifies a function; `environmentstore` stores its environment; and `a` identifies the value returned by the function. Because we want to be able to start processes on remote nodes, we need to be able to transmit closures, and so closures must be serializable. We might be tempted to start by making all functions serializable:

— *wrong*

```
instance Binary (a → b) where
```

```
  put x = — ??
```

```
  get = — ??
```

Unfortunately, we cannot write this instance in Haskell: there is no way to introspect into a function at runtime. That is, we can neither get a serializable function representation, nor a serializable form of its environment, both of which are necessary for creating a closure. However, we can not avoid this problem: if we are to have a function capable of starting new processes on remote nodes, we must be able to construct serializable closures.

One solution to the problem of the serialization of functions is to extend the runtime system, so that any type at all can be serialized. This is the approach taken by most languages that address this issue. Erlang, for example, can transparently produce a closure for an arbitrary function. However, there are problems with this approach:



- Ceding the task of environment serialization to the runtime means that the programmer loses an overview of what data is serialized and when. One might say that it makes serializing data *too easy*. We can imagine a programmer accidentally transmitting a huge chunk of data with an innocent-looking function call, just because that function has a free variable bound to an entire database. If the function is part of a third-party library, the programmer might not even have a way of knowing exactly what is being invisibly serialized as part of its environment. While that is acceptable in a strictly shared-memory arrangement, in distributed applications, the cost of transmitting data is of paramount concern, and the programmer should retain control of it.
- Some data types should never be serialized. As mentioned in Section 2.1.1, data types specific to shared-memory concurrency should never be sent to a remote system. In Cloud Haskell, we enforce that provision through restrictions on serializations. We can imagine a programmer extending those restrictions to other strictly local data types, such as handles to local resources. Thus, we do not want to leave decisions about serializability entirely up to the runtime.
- Similarly, some data types can be serialized more efficiently with manual intervention, e.g. by stripping redundant structures. Again, the aim is to reduce the cost of transmitting data.

Which functions can be safely serialized without extending the runtime system? Certainly, those without free variables. We include such functions in a category of safely serializable functions which we call *freeless*. We can also assume that functions whose only free variables are parameters are serializable, since those parameters will be serialized with them; thus, they too are freeless. Unfortunately, functions with no free variables are hard to come by: even `(+)` is a value in Haskell, so a function like `add a b = a + b` has a free variable. Fortunately, we know *a priori* that `(+)` is available at the point of deserialization, and therefore does not need to be serialized; we know this, because `(+)` is part of Haskell. So `add` is freeless, after all.

This train of thought leads us to the conclusion that we can serialize *any* value that we know will be available at the point of deserialization. In Cloud Haskell, we require that the same binary be running on all nodes. While this may seem an onerous demand, it means that any top-level name can be safely serialized, since those values are guaranteed to be present. We can extend this thought even further and give a precise definition of freeless: a *freeless* function is one whose only free variables are parameters, top-level names, or are freeless themselves.

How can we identify freeless functions? Rather than allow serialization of all freeless functions, we allow serialization of a subset of them: specifically, those that are top-level

functions. This approximation does not greatly hinder the framework’s flexibility and ensures that the serialization mechanism can be implemented entirely in library code.

We now turn to the issue of serializing a closure’s environment. Because of the restrictions we have placed on serializable functions, we know that the only free variables we need to serialize are a function’s parameters. The `Data.Binary` module gives us an `encode` function that translates a serializable tuple of arguments into a string, and a corresponding `decode` function that reverses that operation:

```
encode :: Binary a => a -> ByteString
decode :: Binary a => ByteString -> a
```

Now the `environmentstore` type in our definition of `Closure` is simply a `ByteString`. The problem with this strategy is in deserialization: Haskell, as a statically typed language, demands that we know the type of data beforehand. What type shall `decode` return, then? Clearly, it must return the type of the arguments of the function to call, but this is something we find out at runtime.

The solution is to require the represented function itself to deserialize its own parameters. That is, for each function that may be called via a `Closure`, either that function must simply accept a `ByteString`, which it gives to `decode`; or there must be a wrapper function that statically knows the type of the underlying function, and calls `decode` on its behalf.

Next, we must find a suitable type to take the place of `functionrep` in the definition of `Closure`. This type must be able to uniquely identify a remotely-callable function. Since we have restricted remotely-callable functions to top-level functions, we have chosen to use a string containing the function’s fully-qualified name (that is, module plus name) as its representation, since all top-level functions must have unique names, unlike locally-defined or lambda functions. The next task is to encode the environment; as already shown, it is sufficient to encode a function’s parameters as a `ByteString`. Thus, the final definition of `Closure` is:

```
data Closure a = Closure String ByteString
```

If we want to run a function of this signature:

```
module Foobar where
someFun :: [Int] -> String -> ProcessM ()
```

Then we can call `spawn` like this:

```
— correct but cumbersome
let theclo = Closure "Foobar.someFun" (encode ([3,4], "zmrzlina"))
in spawn someNode theclo
```

But how can the Spawner process connect the string `"Foobar.someFun"` with the `someFun` function? And wouldn’t it be nice if there were a more concise way of creating closures?

Both of these questions are addressed by Cloud Haskell’s `remotable` function, which generates boilerplate wrappers and lookup tables required for creating and using closures. It does this using the Template Haskell metaprogramming facility. The following Template Haskell splice:

```
$( remotable [ 'someFun ] )
```

will expand at compile time to this:

```
someFun__closure :: [Int] → String → Closure (ProcessM ())
someFun__closure a1 a2 = Closure "Foobar.someFun__impl" (encode (a1, a2))

someFun__impl :: ByteString → ProcessM ()
someFun__impl a =
    do { (a1, a2) ← liftIO (decode a)
        ; someFun a1 a2 }

__remoteCallMetaData :: RemoteCallMetaData
__remoteCallMetaData =
    putReg someFun__impl "Foobar.someFun__impl"
```

In the above expansion, `someFun__closure` is the user-visible closure generator; it is simply a shorthand for the manual closure construction shown in the previous example. The programmer will use `someFun__closure` to create closures suitable for use with `spawn`. Note that `someFun__closure` returns a `Closure (ProcessM ())`, as it is derived from an original function of type `ProcessM ()`; thus, `remotable` provides static guarantees matching the type of closure to the type of the underlying function.

The function directly named by `someFun__closure` is `someFun__impl`; this is because a remotely-invoked function is given a `ByteString` containing its encoded environment, which it will then decode. Since the framework would not otherwise be able to statically know the type of the user-defined function, it generates a wrapper function to deserialize the environment and call the underlying user function.

Finally, we need a way to call `someFun__impl` by name. Haskell does not provide a run-time name lookup service, so we must make one ourselves. To wit, `__remoteCallMetaData` provides a mapping from names to functions. At initialization, this function generates a table of type `Map String Dynamic` that is used by the Spawner when invoking closures; we use Haskell’s `Dynamic` type to wrap functions of arbitrary type. Each module that uses `remotable` has its own `__remoteCallMetaData`, and all such tables must be passed to `remotelnit` when initializing the framework.

In the end, we simply call our function by using syntax very similar to a normal function call:

```
spawn someNode (someFun__closure [3,4] "zmrzlina")
```

Note that because the entire procedure of function lookup and closure invocation is performed in normal type-safe library code, it is impossible for a segmentation fault or other uncontrolled error to result. For example, if due to a version mismatch between nodes, the function expressed in a closure cannot be found, the error will be caught in library code and logged. Similarly, if a closure contains the wrong type of parameters, the decoding function will fail in a controlled way.

# Chapter 5

## Evaluation

### 5.1 Clustering

In addition to the detailed analysis of an example Cloud Haskell application presented in Section 3.1, we now briefly present another example application. This program implements the well-known  $k$ -means algorithm. The algorithm is used to identify natural clusters within sets of data points; its input is a set of data points and an integer  $k$ , and its output is an assignment of each point to one of  $k$  clusters.

The  $k$ -means algorithm is implemented in terms of MapReduce. Each node is assigned a role of either *mapper* or *reducer*; the data points are divided evenly between the mappers. Each mapper then calculates the distance between each of its data points and the (initially random) centroid of each cluster, and on that basis assigns each point to the best cluster. The new assignments are collected by the reducer nodes, which take the average of all points assigned to a given cluster, yielding new cluster centroids. The new centroids are sent back to the mappers and used to repeat the algorithm, until either the centroids stabilize or until the maximum number of iterations is reached.

The  $k$ -means algorithm is computationally demanding, data intensive, and easily partitioned, and so provides a useful test case for Cloud Haskell. The output from successive iterations of the algorithm using a sample data set is shown in Figure 5.1. The source code for two  $k$ -means implementations is included in the distribution: one implementation based on the process layer, and the other based on the task layer. We feel that these two implementation demonstrate the power and flexibility of the Cloud Haskell API, as well as the contrasting elements of the two interfaces.

### 5.2 Performance

One goal of a distributed system is to be able coordinate compute- and data-intensive algorithms among many nodes without incurring a performance overhead. Here we discuss

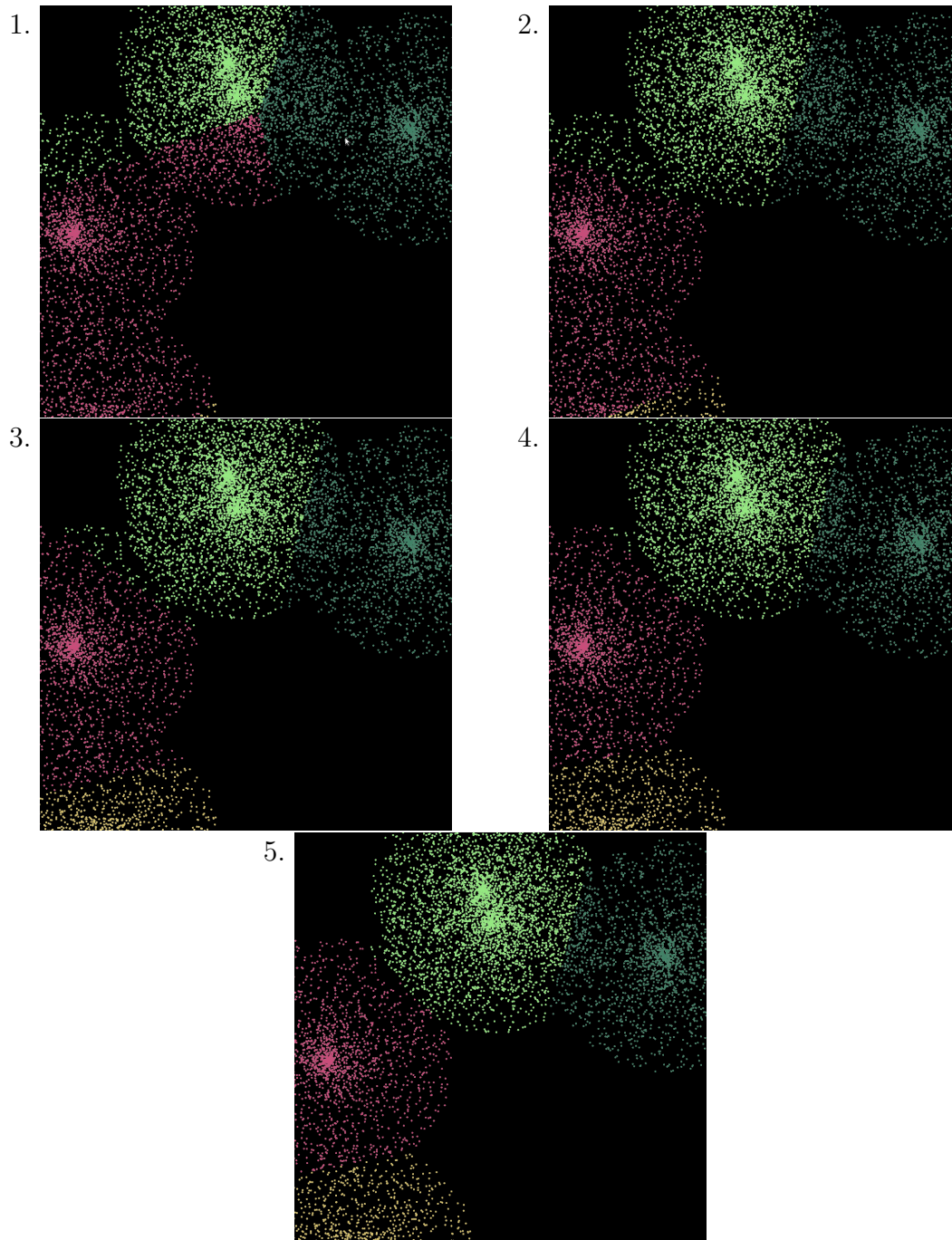


Figure 5.1: The allocation of two-dimensional points to clusters in successive iterations of the  $k$ -means algorithm,  $k = 4$ . Initially, the center point of each cluster is chosen randomly. With each iteration, the clusters move closer to convergence.

the performance of our implementation of the  $k$ -means algorithm, introduced in Section 5.1.

We tested two versions of the  $k$ -means algorithm: the process layer-based Cloud Haskell implementation; and the Apache Mahout implementation for the Hadoop framework. We deployed both implementations of  $k$ -means on an Amazon EC2 cluster. Each virtual machine in the cluster was an Amazon instance of type `m1.small`, which is a single core machine with 1.7 GB of memory, and was running Ubuntu 10.04 (image `ami-311f2b45`).

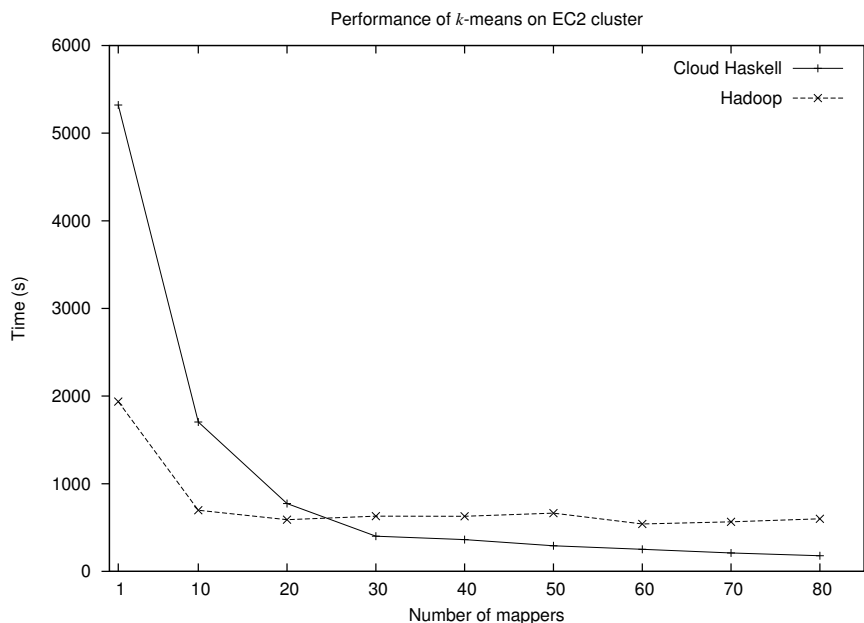


Figure 5.2: The run-time of the  $k$ -means algorithm. We compare the algorithm as implemented with Cloud Haskell’s process layer, and the Mahout implementation running under Hadoop. The input data was one million 100-dimensional data points. We used one reducer nodes for all tests.

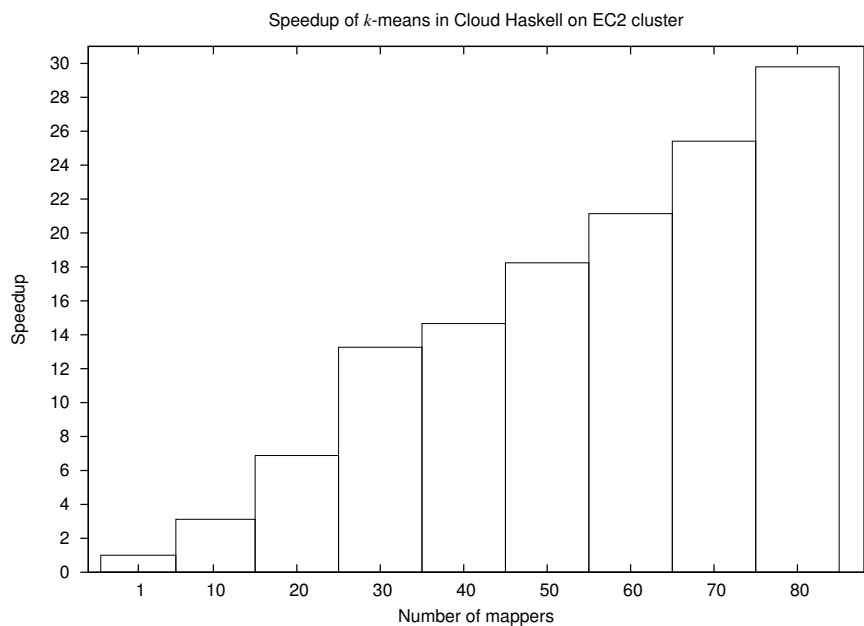


Figure 5.3: The speedup of the  $k$ -means algorithm in Cloud Haskell, compared with execution on a single mapper. The results are based on the same tests from Figure 5.2.

We deployed the Hadoop version using Amazon’s Elastic MapReduce interface, while the Cloud Haskell version was deployed using a hand-written script based on `scp`.

We used as input one million randomly generated, evenly distributed 100-dimensional data points, and extracted  $k = 10$  clusters. The number of iterations was fixed at five. The results of the tests are summarized in Figure 5.2. In Figure, Figure 5.3, we show the speedup achieved by adding additional mapper nodes to a Cloud Haskell deployment.

The results show that Cloud Haskell has performance roughly comparable with that of the Hadoop framework, and in some cases superior. Although the Hadoop implementation performs better with fewer nodes, the Cloud Haskell version overtakes it as more nodes are added, and retains this lead. The greatest bottleneck in Cloud Haskell's performance is acquiring and loading the data; we hope to address this issue with improved file handling in future versions.



# Chapter 6

## Conclusion

The Cloud Haskell project has successfully shown that a distributed programming language can be embedded in a strongly-typed language like Haskell. We believe that Cloud Haskell represents a promising beginning for distributed computing in Haskell. We have shown that Haskell is a good fit for this problem space, thanks to its strong static typing, purity, and monads. We have also shown that Cloud Haskell presents useful abstractions that make it relatively easy to construct complex distributed applications.

Cloud Haskell is already being used in other projects, including a distributed hash table being developed at Heriot-Watt University. It is pleasing to see the interest within the academic community in continuing our work.

### 6.1 Related work

As should be clear, the most direct inspiration for this project comes from Erlang [1], from which we borrow the process-oriented message-passing architecture and fault-monitoring interface. In the scientific computing community, a widely-used solution for distributed parallelism is the Message Passing Interface (MPI) [8]. Unlike Cloud Haskell, MPI aims to be language independent, with interfaces for Fortran, C, C++ and other languages. While language independence makes it more accessible, it also reduces its integration with any single language, whereas many of Cloud Haskell's features are tightly coupled with Haskell's type system.

Our task layer was inspired by the CIEL execution engine [17] and Skywriting language [16] of Murray *et al.* We improve on the CIEL model by allowing programmers to write their entire program in one language, rather than using one language for calculations and another for process coordination. The key feature we have taken from CIEL, futures/promises, has been used in functional languages before, including Scheme [15], and also for distributed programming: Schwendner [21] has developed a distributed Scheme based on futures, although without fault tolerance or message-passing.

This is not the first attempt at a distributed computing framework in Haskell. Glasgow Distributed Haskell [19], though, takes a very different approach from Cloud Haskell, by attempting to reproduce the semantics of shared-memory programming in a distributed environment. We feel that a distributed programming model should reflect the cost model of the underlying hardware, and therefore we present an interface very different from traditional concurrent programming. In particular, Cloud Haskell does not provide mutable distributed variables as part of its interface.

MapReduce [5], with its open-source implementation Hadoop<sup>1</sup>, is the best-known distributed computing framework, but suffers from rigidity. In particular, it is insufficiently expressive for writing iterative algorithms. Dryad [11] improves on MapReduce by allowing the data flow of an algorithm to be expressed as a directed acyclic graph, but in both frameworks, the data flow must be fully specified before computation begins. The need for data-dependent control flow was one of the motivations for CIEL.

Haskell’s traditional shared-memory concurrency mechanism is known as Concurrent Haskell [13], and is based on lightweight threads. Threads communicate through mutable variables or channels, which are similar to Cloud Haskell’s distributed channels. Haskell has also been extended for parallel vectorization, in the form of Data Parallel Haskell [4].

The Go language<sup>2</sup> and Concurrent ML [20] have both been designed with concurrency in mind, although they do not specifically address distributed computing. Like Concurrent Haskell, they use lightweight processes and communication over typed channels.

Java has its own system for executing functions on remote computers: Remote Method Invocation (RMI) [18]. Although its serialization mechanism is built into the language through introspection, it gives the programmer a high degree of control: the programmer must manually indicate which data types are serializable and may override the serialization protocol for particular types. It thus avoids the usual disadvantages of automatic serialization.

CORBA provides the Interface Description Language (IDL) [25] to specify language-independent remotely-callable services. Web services often use the SOAP standard to encode remote procedure calls. However, neither of these systems deal with the problem of transmitting functions that capture their environment.

The Emerald language [14] is an object-oriented language with mobile objects, i.e., its objects can be serialized and migrated to other systems. Because its stack frames are objects as well, functions that capture their environment can be automatically migrated.

Acute [22] and HashCaml [2] are designed with type-safe serialization in mind. The attach type representations to serialized data, ensuring safety even if the structure of a type changes between versions.

---

<sup>1</sup><http://hadoop.apache.org/>

<sup>2</sup><http://golang.org/>

The Clean [24] language also addresses the problem of function serialization, and in fact allows data of arbitrary type to be transmitted between programs, even if those programs do not have all their types in common. The mechanism that it uses to accomplish this, an external type library written at compile time, requires extensive integration with the language and is not directly adaptable to a distributed environment, but suggests interesting avenues of research.

## 6.2 Future work

In this section, we discuss some ideas for future directions that work with Cloud Haskell could take.

Experience building applications with Cloud Haskell shows that the mostly sorely needed feature is integration with a distributed file system. This absence is most felt when programming with the task layer, where the only way to move data is by promises, which have the disadvantage of being slow: creating a promise requires coordination with the master node, and dereferencing a large number of promises means that a worker node may spend a lot of time waiting, since there is no anticipatory pre-fetching of promise values. Additionally, our serialization system is slow when dealing with huge blocks of data. It would be useful to integrate an existing file system, such as the Hadoop Distributed File System used by Skywriting.

Another problem in Cloud Haskell’s task layer is the fragility of the master node. Currently, the task layer has no provision for redundancy of the master node, meaning that if that node fails, the whole application must restart. In a real-world production environment, this is clearly unacceptable. Production-ready distributed systems usually provide a redundant master node. Even better, a future version of Cloud Haskell could eliminate the master node entirely, and make communication between nodes strictly peer-to-peer.

The task layer’s allocation algorithm is fairly naive. Each new task is assigned to a node in a round-robin fashion, disregarding the actual capabilities of a node and the requirements of the task. It would be worthwhile to develop a more advanced scheduler that could monitor the actual processor and memory load of nodes. The scheduler might also consider the dependencies of tasks, as does Quincy [12].

Cloud Haskell could benefit from dynamic software updates [10], as Erlang does. So, when a new version of code is released, it could be transmitted to every host in the network, where it will replace the old version, without having to restart the application. We decided not to go in this direction with our framework, partly because code update is a problem that can be separated from other aspects of building a distributed computing framework, and partly because solving it is especially difficult in a language that compiles to machine code, as does Haskell. Erlang, on the other hand, uses a bytecode interpreter and so retains more control over loading and execution.

A consequence of not supporting over-the-wire code updates is that upgrading a running distributed application requires stopping, upgrading, and restarting all nodes, a potentially time-consuming and error-prone procedure. One problem that can arise during an upgrade is type inconsistency between versions, when the structure of a message changes. Depending on the change made, a message might fail to deserialize, resulting in an error message. Even worse, type changes might not be detected by the deserializer, resulting in silently mangled values. The solution is to include a *type fingerprint*, containing a hash of the representation of the encoded type, with each message.

Type ambiguity arises not only between versions, but also between disparate systems. Cloud Haskell currently has no provision for making sure that an `Int` on one machine is the same as an `Int` on another. If a programmer intends to build an application for deployment on a heterogeneous network, then numeric types of explicit width should be used in messages, such as `Int32`, in place of the platform-dependent `Int`. Message encoding should also take into account endian-ness.

Well-known algorithms, such as distributed consensus [6] could be used to assign roles to nodes dynamically. Virtual synchrony [3] could be used to achieve a single view of the shape of the network, rather than let individual nodes reach their own determination about the availability of their peers, which can lead to confusing results in the case of a network bisection.

The message serialization mechanism we have chosen, `Data.Binary`, has limitations. In particular, some user-defined data types cannot be serialized, such as some existential data types and GADTs. Also, recursive data structures, such as circular linked lists, require special attention during serialization. Future work might look at ways to generalize message serialization to these types.

Our method of closure serialization prevents some function types from being invoked remotely. For example, it is impossible to call a function like `reverse :: [a] → [a]`, because the remote side cannot infer from its type how to deserialize its parameters. In Cloud Haskell, polymorphic functions cannot be called through closures. An improvement in the closure mechanism might change this.

# Appendix A

## Configuration reference

Each configuration option can be set in two ways: on the command line of a Cloud Haskell executable, or in a configuration file. The name and location of the configuration file, if used, is determined by the application, but is usually `config`, located in the current directory. The location of the configuration file can also be specified by the `RH_CONFIG` environment variable. If an option is specified both on the command line and in the configuration file, the value given on the command line takes precedence.

The syntax for setting a configuration option differs slightly depending on where you are setting it. On the command line, use a hyphen to introduce the option, followed by the name of the option, then an equals sign, and finally, the value, enclosed in quotation marks if necessary:

```
./MyApplication -cfgRole=SOMEROLE
```

When setting a configuration option in the configuration file, give each setting on a separate line, where each line consists of the name of the option and its value, separated by whitespace. Quoting the value is not necessary. Lines prefixed with a hash (`#`) are comments:

```
# My configuration file
cfgRole SOMEROLE
```

Here is a summary of the user-visible configuration options.

Name	Default	Description
<code>cfgRole</code>	<code>NODE</code>	The user-assigned role of this node determines what its initial behavior is and how it presents itself to its peers.

<code>cfgKnownHosts</code>		A whitespace-delimited list of hosts where nodes may be running. You may specify each host by name or IP address. The <code>getPeers</code> function will query the node registry running on these hosts when returning information about known nodes.
<code>cfgHostName</code>		The hostname, used as a basis for creating the name of the node. If unspecified, the OS will be queried. Since the hostname is part of the node's name, the computer must be accessible to other nodes using this name.
<code>cfgListenPort</code>	0	The TCP port on which to listen to for new incoming messages. If 0, the framework will let the operating system assign a port.
<code>cfgLocalRegistryListenPort</code>	38813	The TCP port for communication with the local node registry. This value must be common among all nodes in your application's network.
<code>cfgPeerDiscoveryPort</code>	38813	The UDP port on which local peer discovery broadcasts are sent and received. If 0, peer discovery broadcasts are disabled, and so only nodes on hosts listed in <code>cfgKnownHosts</code> will be found. This value must be common among all nodes in your application's network.
<code>cfgNetworkMagic</code>	MAGIC	The unique identifying string for this network or application. The value must not contain spaces. The uniqueness of this string ensures that multiple applications running on the same physical network won't accidentally communicate with each other.
<code>cfgRoundtripTimeout</code>	10000000	Time in microseconds to wait for a response from a system service on a remote node. If the network has high latency or congestion, it may be necessary to increase this value to avoid incorrect reports of node inaccessibility.
<code>cfgMaxOutgoing</code>	50	A limit on the number of simultaneous outgoing connections per node.

---

<code>cfgPromiseFlushDelay</code>	<code>5000000</code>	Time in microseconds before an promise's value is flushed to disk. If 0, values are never flushed to disk.
<code>cfgPromisePrefix</code>	<code>rpromise-</code>	Prepended to the filename of flushed promise values. This value may contain a full or relative pathname.

# Bibliography

- [1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in Erlang, 1993.
- [2] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-safe distributed programming for Ocaml. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 20–31, New York, NY, USA, 2006. ACM.
- [3] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21:123–138, November 1987.
- [4] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.
- [5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
- [6] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34:77–97, January 1987.
- [7] J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, and R. Barga. DryadLinq for scientific analyses. *e-Science and Grid Computing, International Conference on*, 0:329–336, 2009.
- [8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.
- [9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '05, pages 48–60, New York, NY, USA, 2005. ACM.
- [10] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. *SIGPLAN Not.*, 36:13–23, May 2001.



- [11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.
- [12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.
- [13] S. P. Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, pages 267–305, Berlin, Heidelberg, 2009. Springer-Verlag.
- [14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *TOCS*, 6(1):109–133, 1988.
- [15] L. Moreau. The semantics of Scheme with future. *SIGPLAN Not.*, 31:146–156, June 1996.
- [16] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.
- [17] D. G. Murray and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI '11: Proceedings of the eighth symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011. USENIX.
- [18] E. Pitt and K. McNiff. *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [19] R. Pointon, P. Trinder, and H.-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In M. Mohnen and P. Koopman, editors, *Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 53–70. Springer Berlin / Heidelberg, 2001.
- [20] J. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.
- [21] A. Schwendner. Distributed functional programming in Scheme. Master's thesis, Massachusetts Institute of Technology, 2010.
- [22] P. Sewell, J. Leifer, K. Wansbrough, F. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: high level programming language design for distributed computation. *Journal of Functional Programming*, 17(4–5), 2007.
- [23] T. Sheard and S. P. Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

- [24] T. van Noort, P. Achten, and R. Plasmeijer. Ad-hoc polymorphism and dynamic typing in a statically typed functional language. In *Proc 6th ACM Workshop on Generic Programming, Baltimore*. ACM, Sept. 2010.
- [25] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, Feb. 1997.