

# Distributed Storage with CloudHaskell

Julian Porter  
julian.porter@lincoln.oxon.org

## Abstract

MapReduce is an extremely popular paradigm for distributed computing. In this note we describe a necessary component to a distributed MapReduce implementation in Haskell, that is, a simple distributed storage service built with CloudHaskell. A working implementation may be found at [2]. This will form the basis for a full distributed MapReduce demonstrator, to be discussed in a future paper.

## 1 Introduction

In [6] we described a monadic approach to generalising MapReduce in Haskell, demonstrating an implementation capable of parallelism on a single machine and stated a goal of running on a distributed cluster. In this note we describe the first step on the way to that goal, which is a simple distributed storage service, capable of handing input data to processing nodes in the cluster and collecting their results. Our implementation is a proof-of-concept and uses a simple in-memory store, but the generalisation to a database or other similar store is obvious.

Our approach is also notable as a practical application of the CloudHaskell distributed computing package (see [3]). We discuss how Haskell's strict type system, has forced design decisions. At first this restriction, while comes down to storing all data in the form of a `[ByteString]`, may seem limiting, but we will show that it is easy to abstract. Moreover, if we want to use (say) a relational database as a persistent back-end store, then we would have to do something of the sort anyway, so the apparent limitation, as usual, turn out to be strengths.

This note is organised as follows. First we give a brief theoretical discussion of storage services in Haskell, and then describe how a distributed storage service may be used in distributed applications in order to motivate a specification. Then we go on to discuss its implementation, with particular reference to types and restrictions imposed by CloudHaskell. Finally we discuss next steps.

### 1.1 Sample code

Sample code is available at `git://github.com/Julianporter/Distributed-Haskell.git`. It depends on CloudHaskell, which may be found at `git://github.com/jepst/CloudHaskell.git`.

## 2 The service

### 2.1 The storage monad

At its simplest a store is something that allows us to put and get data, so:

**Definition 2.1.1.** *A parametrised type  $s$  is a **store** if, for any type  $a$  in some class `Storable` there are operations:*

$$\begin{aligned} \text{get} &:: (\text{Storable } a) \Rightarrow s\ a \rightarrow a \\ \text{put} &:: (\text{Storable } a) \Rightarrow a \rightarrow s\ a \end{aligned}$$

*subject to the constraints that  $\text{get} \circ \text{put} = \text{id}$  and  $\text{put} \circ \text{get} = \text{id}$ .*

The constraints just say that we get back what we put in. This is quite a rewarding type to study in its own right. We can prove:

**Proposition 2.1.2.** *If  $s$  is a store then it is a monad, with*

$$\begin{aligned} x \gg f &:= f \text{ (get } x) \\ \text{return} &:= \text{put} \end{aligned}$$

*Proof.* We need to prove that  $s$  obeys the monad laws, which may be found in [4]. Expanding the first law  $\text{return } a \gg f \equiv f a$  gives

$$\begin{aligned} \text{return } a \gg f &\equiv \text{put } a \gg f \\ &\equiv f \$ \text{get (put } a) \\ &\equiv f a \end{aligned}$$

In the second law  $x \gg \text{return} \equiv x$  we get

$$\begin{aligned} x \gg \text{return} &\equiv x \gg \text{put} \\ &\equiv \text{put } \$ \text{get } x \\ &\equiv x \end{aligned}$$

And finally, the third law  $s \gg (\lambda x \rightarrow f x \gg g) \equiv (s \gg f) \gg g$  gives

$$\begin{aligned} y \gg (\lambda x \rightarrow f x \gg g) &\equiv y \gg (\lambda x \rightarrow g \$ \text{get (f } x)) \\ &\equiv (\lambda x \rightarrow g \$ \text{get (f } x)) \$ \text{get } y \\ &\equiv g \$ \text{get (f } \$ \text{get } y) \\ &\equiv g \$ \text{get (y } \gg f) \\ &\equiv (y \gg f) \gg g \end{aligned}$$

□

We can, in fact, go further, and see that a store respects monoidal structure in the data being stored:

**Proposition 2.1.3.** *If  $s$  is a store and  $m$  is a Storable monoid, then  $s m$  is a monoid with:*

$$\begin{aligned} \square &:= \text{put } \odot \\ x \boxplus y &:= \text{put } \$ (\text{get } x) \oplus (\text{get } y) \end{aligned}$$

where we use the notations  $\odot$  and  $\square$  for the monoidal zero in  $m$  and  $s m$  respectively, and similarly  $\oplus$  and  $\boxplus$  for monoidal addition.

*Proof.* We need to verify the monoid laws, which are to be found in [1]. Expanding the first law  $\square \boxplus s = s$  we get

$$\begin{aligned} \square \boxplus x &\equiv \text{put } \$ (\text{get } \square) \oplus (\text{get } x) \\ &\equiv \text{put } \$ \odot \oplus (\text{get } x) \\ &\equiv \text{put } \$ \text{get } x \\ &\equiv x \end{aligned}$$

The proof of the second law  $x \boxplus \square = x$  follows by a trivial modification of that for the first law. Turning to the third law  $s \boxplus (s' \boxplus s'') = (s \boxplus s') \boxplus s''$  we get

$$\begin{aligned}
x \boxplus (y \boxplus z) &\equiv x \boxplus (\text{put } \$ (\text{get } y) \oplus (\text{get } z)) \\
&\equiv \text{put } \$ (\text{get } x) \oplus \text{get } (\text{put } \$ (\text{get } y) \oplus (\text{get } z)) \\
&\equiv \text{put } \$ (\text{get } x) \oplus \$ (\text{get } y) \oplus (\text{get } z) \\
&\equiv \text{put } \$ ((\text{get } x) \oplus (\text{get } y)) \oplus (\text{get } z) \\
&\equiv \text{put } \$ \text{get } (\text{put } \$ (\text{get } x) \oplus (\text{get } y)) \oplus (\text{get } z) \\
&\equiv \text{put } \$ (\text{put } \$ (\text{get } x) \oplus (\text{get } y)) \boxplus z \\
&\equiv (x \boxplus y) \boxplus z
\end{aligned}$$

□

In fact we can show that if  $m$  is a *Storable* additive monad (see [5]) then so is  $s\ m$ . However, the proof would take us too far afield, and so will be held back for a future paper.

What this means is that if we want to store a data type which has an additive structure, like say lists with  $\odot := []$  and  $\oplus := ++$ , then we can concatenate data either locally or in the store and get the same results. To make this precise:

**Lemma 2.1.4.** *If  $s$  is a store and  $m$  is a Storable monoid, then*

$$(\text{put } x) \boxplus (\text{put } y) \equiv \text{put } (x \oplus y)$$

*Proof.* Trivially:

$$\begin{aligned}
(\text{put } x) \boxplus (\text{put } y) &\equiv \text{put } \$ (\text{get } \$ \text{put } x) \oplus (\text{get } \$ \text{put } y) \\
&\equiv \text{put } (x \oplus y)
\end{aligned}$$

□

So this trivial result shows that if the data being stored is monoidal, then *any* store reflects the monoidal structure of the data.

## 2.2 Our requirement

So, what do we want of a storage service? For distributed computing there are three main applications of storage: passing data to processing nodes, collecting data output by the processing nodes, and concatenating these outputs to form the input for the next round of processing. We want to do this in a simple client-server paradigm where we have one **master node**, which manages the process, one **storage node**, which runs the storage service, and a number of **processing nodes**.

We can model this as follows. Assume that the data is in some *Storable* monoid  $m$  and  $s$  is a store. The storage node runs two stores *inputs* and *outputs*, both of type  $s$ . Denote the service running on the storage node by  $S\ m\ a\ b$  where  $m\ a$  is the type of the data in *inputs* and  $m\ b$  that of the data in *outputs*. Then we need three operations:

- $\text{push} :: S\ m\ a\ b \rightarrow m\ a \rightarrow S\ m\ a\ b$  appends a value of type  $m\ a$  to the value stored in *inputs*. So  $\text{push}$  applied to  $x$  leaves *outputs* unchanged, and sends *inputs* to  $\text{inputs} \boxplus (\text{put } x)$ .
- $\text{pull} :: S\ m\ a\ b \rightarrow m\ b$  returns the value stored in *outputs*, so it is simply  $\text{get}$  applied to *outputs*.
- $\text{exchange} :: S\ m\ a\ b \rightarrow S\ m\ b\ c$  puts the value stored in *inputs* into *outputs* and sets *inputs* to  $\square$ .

To walk through this, the master initialises processing by using  $\text{push}$  to load initial data and then applies  $\text{exchange}$  to swap it from *inputs* to *outputs*. Then each round of processing involves each processing node using  $\text{pull}$  to get its input, then using  $\text{push}$  to return its output. When they are all finished the master swaps *inputs* to *outputs* with  $\text{exchange}$  and the next round begins.

The observant reader will have noticed that this is fraught with type-related complications. This will turn out to be extremely significant when we turn to implementation.

### 3 Implementation

#### 3.1 A little CloudHaskell

We describe on the fragment of CloudHaskell that we use in this application; those wishing to know more should consult [3].

The aspect of CloudHaskell that we make use of is simple message passing, where the server node acts as a message consumer, while client nodes send it messages to which it may reply. It is, essentially, a Haskell equivalent of TCP/IP client-server sessions. CloudHaskell requires that all types of data that pass between nodes in the cluster are in the class *Serializable*, which means that they are *Typeable* and *Binary*. GHC allows us to derive *Typeable* automatically, but we have to write some not very-illuminating serialisation / deserialisation boiler-plate to implement *Binary*. As this is not significant for our discussion, we shall take it as read that all data types implement *Serializable*.

All message-passing happens in the *ProcessM* monad, which can be thought of as a distributed version of *IO*. Every node in the cluster has a unique identifier of type *ProcessID*. The message-passing API then consists of two methods:

$$\begin{aligned} \text{send} &:: (\text{Serializable } a) \Rightarrow \text{ProcessID} \rightarrow a \rightarrow \text{ProcessM } () \\ \text{expect} &:: (\text{Serializable } a) \Rightarrow \text{ProcessM } a \end{aligned}$$

We send a value  $x$  to the node with unique identifier  $p :: \text{ProcessID}$  by invoking

$$\text{send } p \ x$$

To receive messages a node invokes *expect*. This puts it into a sleep state, in which it remains until an incoming message addressed to it is received, at which point it wakes and returns the message wrapped in *ProcessM* (this is just like a standard TCP socket). So if

$$f :: (\text{Serializable } a) \Rightarrow a \rightarrow \text{ProcessM } ()$$

is some function that we use to process the incoming message, we invoke

$$\text{expect} \gg\equiv f$$

within *ProcessM*.

These communications channels are typed, so if the server is running  $\text{expect} \gg\equiv f$  as above, and a client sends a message of type  $b$  to the server's unique identifier, the server will ignore it. This becomes crucially important below.

#### 3.2 Type issues

We start by defining the messages that pass between client and server. Either we want to push data into the server, pull some data from it, or exchange the input and output stores, so we have:

$$\mathbf{data} (\text{Serializable } a) \Rightarrow \text{DataMessage } a = \text{Push } [a] \mid \text{Pull} \mid \text{Exchange}$$

Now we might assume we could implement the server as something like this:

```

1  getData (Serializable a, Serializable b) => :: [a] -> [b] -> ProcessM ()
2  getData inputs outputs = do
3    (pid, m) <- expect

```

```

4  case m of
5    Push x    → do
6              getData (inputs ++ x) outputs
7    Pull      → do
8              send pid outputs
9              getData inputs outputs
10   Exchange → do
11         getData [] inputs

```

*inputs* and *outputs* are the two stores described above, implemented as simple in-memory stores. Note in line 3 that a client passes its *ProcessID* to the server so that the server knows to whom it should reply.

There is a problem with this. What type is [] in line 11? It must be the type of the data that will subsequently be pushed to the server, but we don't yet know what that is. Fortunately, there is a work-around. Redefine *DataMessage* as:

```
data (Serializable a) ⇒ DataMessage a = Push [a] | Pull | Exchange [a]
```

and change the server to:

```

1  getData :: (Serializable a, Serializable b) ⇒ [a] → [b] → ProcessM ()
2  getData inputs outputs = do
3    (pid, m) ← expect
4    case m of
5      Push x    → do
6                getData (inputs ++ x) outputs
7      Pull      → do
8                send pid outputs
9                getData inputs outputs
10     Exchange x → do
11           getData x inputs

```

Now when we initialise the server, we tell it to expect data of type *c*. We might think we could do this with:

```
send p $ (myPID, Exchange ([] :: [c]))
```

Unfortunately, Haskell does not let us use type variables like *c* in function definitions, so instead we end up having to do this:

```

doExchange :: (Serializable a) ⇒ ProcessID → ProcessID → a → ProcessM ()
doExchange myPID yourPID x = send yourPID (myPID, Exchange [x])

```

and change *getData x inputs* to *getData (tail [head x]) inputs* in line 11 of the server<sup>1</sup>.

With these modifications the code will compile. Recall that the message channels used by *send* and *expect* are typed, and *expect* ignores messages of the wrong type. Consider an *Exchange* message. When a client sends (*myPID*, *Exchange* [*y*]) to the server, this has the type (*ProcessID*, [*c*]) where *y* :: *c*. But at the server, the type of *x* in lines 10 and 11 is entirely unspecified, so the server will ignore the message *unless* *inputs* :: *c* also. But this means that the storage service can only ever handle data of one type. And

---

<sup>1</sup>If *x* :: *a* then *tail [head x]* ≡ [] :: [*a*].

it gets worse, because in lines 8 and 9, the server knows that it is expecting a *DataMessage*  $c$  for some  $c$ , which is unspecified, and so it ends up ignoring all *Pull* messages. This is unworkable.

Fortunately, there is a very simple solution. What we need is to make *DataMessage* unparametrised, so all data passing to and from the server is of a single type. At first sight this seems unacceptable, as it is a characteristic of MapReduce and other such algorithms that the type of the data being processed changes from round to round. But CloudHaskell requires that all data is *Serializable*, which means that it is *Binary*, which means that it can be converted to and from a *ByteString* with the functions *encode* and *decode* respectively. So the server stores only *ByteStrings*, so all messages are of one type, while the client converts data to / from *ByteStrings* before / after interacting with the server. This removes all of the issues explored above, at the minimal expense of inserting serialisation / deserialisation code into the client API.

### 3.3 The working service

So this is what we do. The implementation falls naturally into three parts: messages, the server, and a client API.

#### 3.3.1 Messages

Define the messages as a simple algebraic type:

```
data DataMessage = Push [ByteString] | Pull | Exchange
```

As discussed above, all data is passed as lists of *ByteStrings*. Conversion to and from types usable by the application is handled by the client API.

#### 3.3.2 The server

The server is:

```
getData :: [ByteString] → [ByteString] → ProcessM ()
getData inputs outputs = do
  (pid,m) ← expect
  case m of
    Push x    → do
      getData (inputs ++ x) outputs
    Pull      → do
      send pid outputs
      getData inputs outputs
    Exchange → do
      getData [] inputs
```

This is a simplified version of the server discussed above, with all data being of type *[ByteString]*. As we might expect, this makes the service extremely robust.

#### 3.3.3 The client API

The client API is:

```
1 putToStore :: (Binary a) ⇒ ProcessId → ProcessId → [a] → ProcessM ()
2 putToStore myPid slavePid xs = do
```

```

3  send slavePid (myPid, Push (encode ($) xs))
4  getFromStore :: (Binary a) => ProcessId -> ProcessId -> ProcessM [a]
5  getFromStore myPid slavePid = do
6    send slavePid (myPid, Pull :: DataMessage)
7    kv ← expect
8    return $ decode ($) kv
9  updateStore :: ProcessId -> ProcessId -> ProcessM ()
10 updateStore myPid slavePid = do
11  send slavePid (myPid, Exchange :: DataMessage)

```

The important parts are in lines 3 and 8. In line 3 we convert  $xs :: [a]$  to a  $[ByteString]$  with the  $encode :: a \rightarrow ByteString$ . Then in line 8 we convert a  $[ByteString]$  to an  $[a]$  with  $decode :: ByteString \rightarrow a$ , which is the inverse of  $encode$ . So we convert data to *ByteStrings* before pushing to the server, and back from *ByteStrings* after pulling from the server.

Now everything is type safe: the type  $a$  in  $putToStore$  and  $getFromStore$  is inferred from the context of the function calling them, so provided consumers match producers, all will be well. In other words, there is no need for application code to know anything about how the storage service works, which is precisely what we want.

## 4 Future directions

Having established the principle that we can build a simple distributed storage service with CloudHaskell, we should replace its persistence mechanism with something more robust than the current in-memory store. An obvious approach would be to use a SQL database, storing  $[ByteString]$ s as BLOBs. We do not see this as being especially problematic, and it is the obvious next step for our work once we have completed the proof-of-concept for MapReduce as a whole.

Note also that our approach allows the storage service to become truly polymorphic in that, provided we have a way to uniquely *decode* each type stored, there is no reason why all the data items stored should be of the same type. All that is required is that they be *Serializable*. This would require a more complex client API, but it opens up intriguing possibilities.

## References

- [1] *Data.Monoid module definition*. URL: <http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Monoid.html>.
- [2] *Distributed MapReduce project GIT repository*. URL: <git://github.com/Julianporter/Distributed-Haskell.git>.
- [3] Andrew P Black & Simon Peyton-Jones Jeff Epstein. “Towards Haskell in the cloud”. In: (2011). URL: <http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf>.
- [4] *Monad page on Haskell wiki*. URL: <http://www.haskell.org/haskellwiki/Monad>.
- [5] *MonadPlus page on Haskell wiki*. URL: <http://www.haskell.org/haskellwiki/MonadPlus>.
- [6] Julian Porter. “MapReduce as a Monad”. In: *The Monad Reader* 18 (2011), pp. 5–16. URL: <http://themonadreader.files.wordpress.com/2011/07/issue18.pdf>.