Data Structures and Functional Programming	**Problem Set**
CS 3110, Spring 2014	
Version:	**Last Modified: March 24, 2014**

Overview

Many modern computational problems require processing very large data sets. Google's MapReduce framework is a platform for distributing programs that handle big data across a large number of computers.

In this problem set, you will implement a simplified version of the MapReduce framework. You will also implement various applications that make use of your framework to process large data sets.

Objectives

This assignment is designed to help you learn the following skills:

- Writing asynchronous programs in an event-driven style
- Developing distributed systems
- Working with the MapReduce paradigm
- Developing software with a partner

This assignment also has a small component on writing formal proofs.

Recommended Reading

- We have provided documentation for a subset of the Async library that should be sufficient for this assignment: **[TODO — link]**. Async contains a very large number of libraries and functions. While you are free to use anything from the full Async library, sticking to the subset we have documented will help you focus.
- Real World OCaml, chapter 18 **[TODO — link]**
- Lectures 17 and 18 **[TODO — link]**
- Recitations **[TODO — fix]**

Part One: Async warmup

These exercises are intended to get you used to asynchronous computation and the Async programming environment.

[TODO — we should probably drop these into a single .ml file]

[TODO — these are just ideas. I'm happy to add/cut/modify any of them. They should be reordered]

Exercise 1:

Implement the asynchronous queue interface defined in `aQueue.mli`. We recommend that you use the `Async.Std.Pipe` module.

Exercise 2:

Write a function `fork` that takes a deferred and two blocking functions, and runs the functions when the deferred becomes determined:

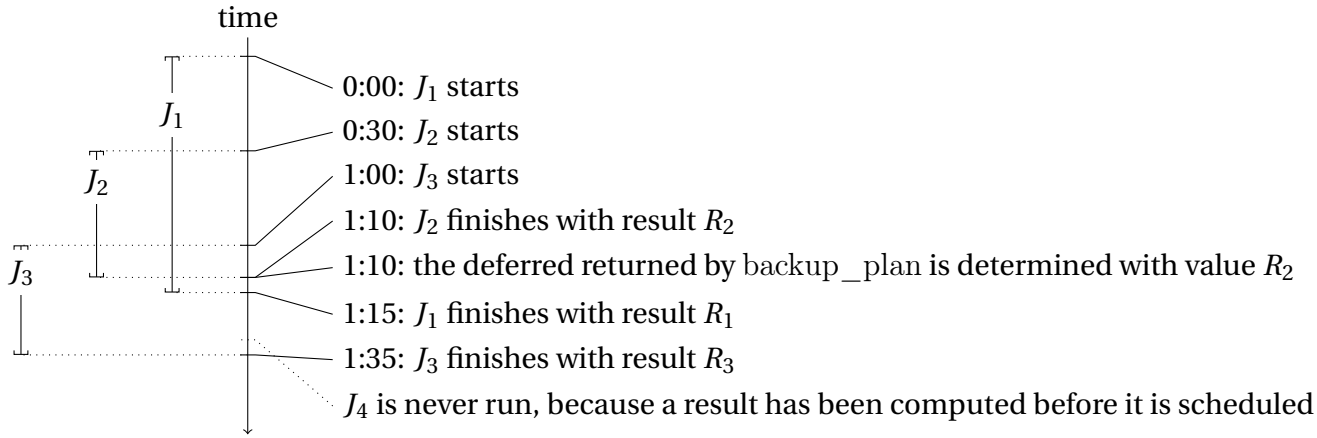
```
val fork : 'a Deferred.t -> ('a -> 'b Deferred.t)
                         -> ('a -> 'c Deferred.t) -> unit
```

The output of the two blocking functions should be ignored.

Exercise 3:

Write a function `backup_plan` that takes a list of blocking functions and tries them one at a time, with a 30 second delay in between. It should return a `Deferred.t` that becomes determined when the first of the jobs finishes. Once any of the jobs have been successfully completed, no new jobs should be started.

For example, suppose that we have jobs `j1`, `j2`, `j3`, and `j4`. Suppose that `j1` takes 1:15. `j2` takes 40 seconds, `j3` takes 35 seconds, and `j4` takes 5 seconds. Calling `backup_plan [j1; j2; j3; j4]` would initiate the following sequence of events:



Exercise 4:

Implement the FilterReader module. An 'a FilterReader.t wraps an 'a Pipe.Reader.t and allows programs to wait for data that matches a predicate to be read from the pipe.

[TODO — this doesn't quite compile(!) because FilterReader returns [`Ok | `Eof] **see examples.ml.]**

For example:

```
# let (r,w) = Pipe.create ();;
# let filterer = FilterReader.create r;;

# FilterReader.read filterer ((>) 5)
  >>| fun x -> printf "reader 1: %i\n" x;;
# FilterReader.read filterer ((>) 3)
  >>| fun x -> printf "reader 2: %i\n" x;;
# FilterReader.read filterer ((>) 8)
  >>| fun x -> printf "reader 3: %i\n" x;;

# Pipe.write w 3;; (* no readers *)
- : TODO = TODO
# Pipe.write w 4;; (* reader 1 doesn't accept, so reader 2 gets it *)
reader 2: 4
- : TODO = TODO
# Pipe.write w 4;; (* no readers --- reader 2 is done *)
- : TODO = TODO
# Pipe.write w 10;; (* reader 1 and reader 3 could handle, but reader 1 came first. *)
reader 1: 10
- : TODO = TODO
# Pipe.write w 10;; (* reader 1 is gone so reader 3 gets it *)
reader 3: 10
- : TODO = TODO
```

Exercise 5:

Using only (>>=), return, and the ordinary List module functions, implement a function with the same specification as Deferred.List.map:

```
val deferred_map : 'a list -> ('a -> 'b Deferred.t) -> 'b list Deferred.t
```

This function should take a list l and a blocking function f, and should apply f in parallel to each element of l. When all of the calls to f are complete, deferred_map should return a list containing all of their values.

Part Two: MapReduce Framework

Modern applications rely on the ability to manipulate massive data sets in an efficient manner. One technique for handling large data sets is to distribute storage and computation across many computers. Google's MapReduce is a computational framework that applies functional programming techniques to parallelize applications.

MapReduce Overview

MapReduce was spawned from the observation that a wide variety of applications can be structured into a **map phase**, which transforms independent data points, and a **reduce phase** which combines the transformed data in a meaningful way. This is a very natural generalization of the `List.fold` function with which you are intimately familiar.

MapReduce jobs provide the code to run in these two phases by implementing the `MapReduce.Job` interface. A MapReduce Job is executed as follows:

- The `map` function takes a single `input` and transforms the value into a collection of intermediate key, value pairs. The `map` function is called once for each element of the input list.

```
val map : input -> (key * inter) list Deferred.t
```

- The `map` results are then aggregated in a `combine` phase. Values associated with the same key are merged into a single list.
- Finally, the `reduce` function takes a `key` and an `inter list` to compute the output corresponding to that key. `reduce` is called once for each independent key.

```
val reduce : key * inter list -> output Deferred.t
```

The advantage of this design is that each call to `map` or `reduce` is independent, so the work can be distributed across a large number of machines. This allows MapReduce applications to process very large data sets quickly while using limited resources on each individual machine.

An example: Word Count

Figure **??** shows a distributed execution of a word counting application. The input to the application is a list of filenames. During the map phase, the controller sends a `MapRequest` message for each input filename to a mapper. The mapper invokes the `map` function in the `WordCount.Job` module, which reads in the corresponding file and produces a list of `(word, count)` pairs. The mapper then sends these pairs back to the controller.

Once the controller has collected all of the intermediate `(word,count)` pairs, it groups all of the pairs having the same word into a single list, and sends a `ReduceRequest` to a reducer. The reducer invokes the `reduce` function of the `WordCount.Job` module, which simply returns the sum

of all of the counts. The reducer sends this output back to the controller, which collects all of the reduced outputs and returns them to the WordCount application.

[TODO — figure no longer quite matches word count app]

Communication protocol

In our implementation of MapReduce, the Controller communicates with Workers by sending and receiving the messsages defined in the Protocol module.

Messages are strongly typed; a message from the Controller to the Worker will have type WorkerRequest.t; responses have the type WorkerResponse.t. Messages can be sent and received by using the send and receive functions of the corresponding module. For example, the Controller should call WorkerRequest.send to send a request to a worker; the worker will call WorkerRequest.receive to receive it.

The WorkerRequest and WorkerResponse modules are parameterized on the Job, so that the messages can contain data of the types defined by the Job. This means that before the worker can call receive, it needs to know which Job it is running. As soon as the controller establishes a connection to a worker, it should send a single line containing the name of the job. After the job name is sent, the controller should only send WorkerRequest.ts and receive WorkerResponse.ts.

We have provided code in the Worker module's init function that receives the job name and calls Worker.Make with the corresponding module.

Once a connection is established and the job name is sent, the Controller will send some number of WorkerRequest.MapRequest and WorkerRequest.ReduceRequest messages to the worker. The worker will process these messages and send WorkerRequest.MapResult and WorkerRequest.ReduceResult messages respectively. When the job is complete, the controller should close the connection.

Error handling

There are a variety of errors that you will have to consider.

infrastructure failure If the controller is unable to connect to a worker, or if a connection is broken while it is in use, or if the worker misbehaves by sending an inappropriate message, the controller should simply close the connection to the worker and continue processing the job using the remaining workers.

If all of the workers die, the map_reduce function should raise an exception **[TODO — what exception?]**.

If a worker encounters an error when communicating with the controller, it should simply close the connection.

application failure If the application raises an exception while executing the map or reduce functions, then the worker should return a JobFailed message. Upon receiving this message, the controller should cause map_reduce to raise an exception **[TODO — what exception?]**.

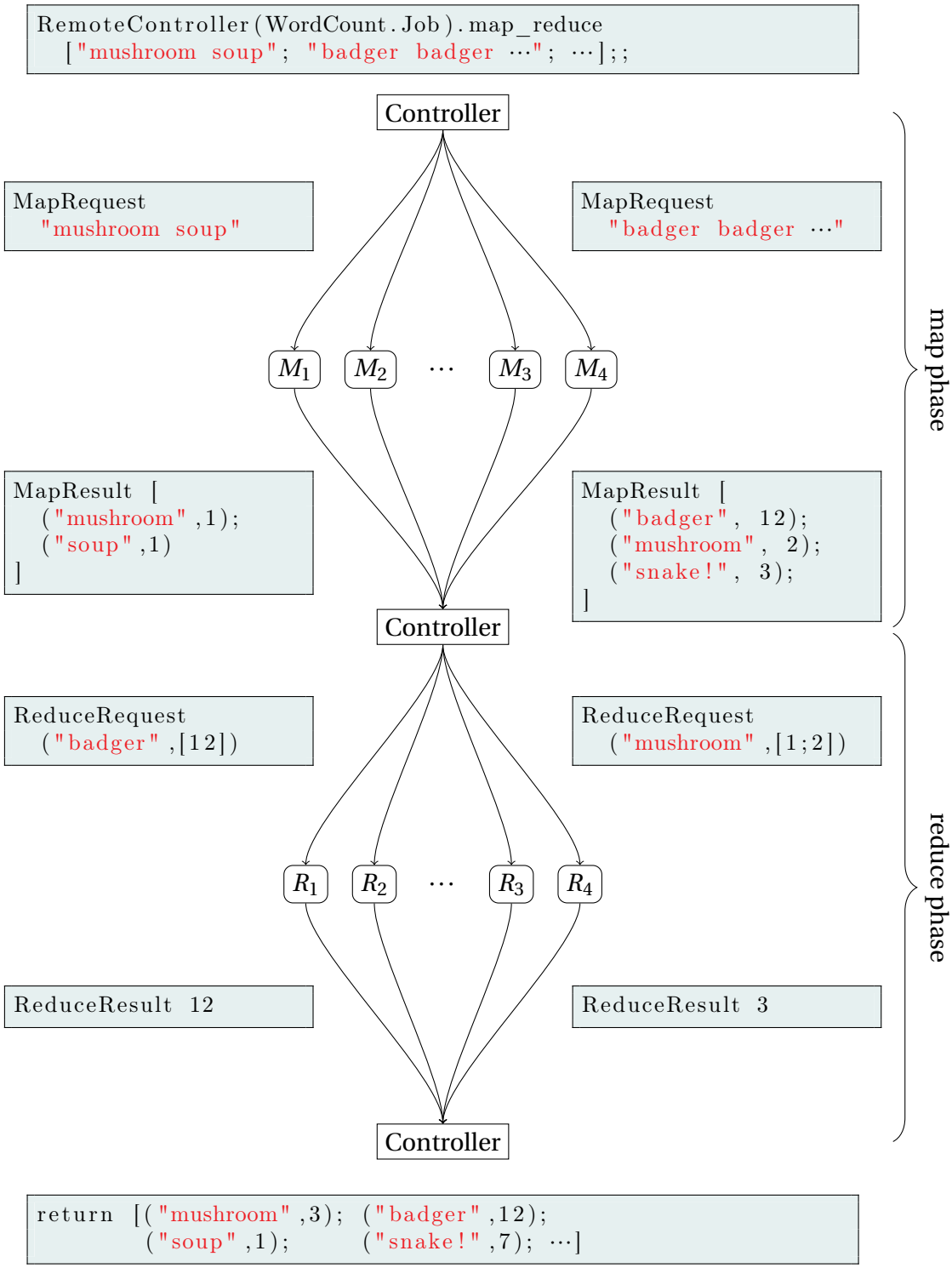


Figure 1: Execution of a WordCount MapReduce job

The name and stack trace of the original exception can be found using the Printexc module from the OCaml standard library; these should be returned to the controller in the JobFailed message, and used to construct the **[TODO — fix]**exception.

slow workers It is possible that some workers may simply be slow. If a worker takes more than **[TODO — time]** processing a single map or reduce request, then the controller should start a second worker to process the same job. This retry process should be repeated: if the second worker takes more than **[TODO — fix]**a third worker should be started and so on. The old workers should not be terminated; the first of these workers to respond should determine the output of the job.

Exercise 6: Implement RemoteController

Implement the RemoteController module. The init function should simply record the provided list of addresses for future invocations of Make.run.

The Make.map_reduce function is responsible for executing the MapReduce job. It should use Tcp.connect to connect to each of the workers that were provided during init. It should then follow the protocol described above to complete the given Job.

You can use the controller to run a given app by running the controllerMain.ml:

```
% cs3110 compile controllerMain.ml
% cs3110 run controllerMain.ml
```

Exercise 7: Implement Worker

Implement the Worker.Make module in the map_reduce directory. The Make.run function should receive messages on the provided Reader.t and respond to them on the provided Writer.t according to the protocol described above.

You can run the worker on a given port by invoking workerMain.ml:

```
% cs3110 compile workerMain.ml
% cs3110 run workerMain.ml 31100
```

The list of addresses and ports that the controller will try to connect to is given in the file addresses.txt.

Part Three: MapReduce Apps

In this part of the assignment you will implement various MapReduce applications.

A MapReduce application implements the MapReduce.App interface, which provides a main function. This function will typically read some input, and then invoke the provided controller with one or more Jobs. We have provided you with the WordCount example application described above to get you started.

We have also provided you with a local controller that you can use to test your apps without a completed implementation of the MapReduce framework. To run an app locally, simply pass the -local option to controllerMain.ml. You should be able to run WordCount out of the box:

```
% cs3110 compile controllerMain.ml
% cs3110 run controllerMain.ml -local SteamWordCount apps/word_count/filenames.txt
```

Exercise 8:

[TODO — fix]

Exercise 9:

[TODO — fix]

Exercise 10:

[TODO — fix]