Data Structures and Functional Programming	**Problem Set 5b**
CS 3110, Spring 2014	**Due Thursday, April 17**
Version: 2	**Last Modified: April 10, 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.

  > **Important:** 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 make the project simpler.

- Real World OCaml, chapter 18
- Lectures 17 and 18, recitation 17.
- Pro Git, chapters 1 and 2

Part One: Software engineering

This problem set is larger than previous problem sets. To help you manage the complexity, you will be required to work with a partner, make use of version control software, and meet with a TA to discuss your approach.

Partners

You are required to work with a partner for this problem set. You and your partner should meet early to jointly discuss your design and division of responsibility.

Each partner is responsible for understanding all parts of the assignment, but your choice of who writes what code is up to you.

Source control

You must use a version control system to manage your development. We recommend `git`, but you are free to use another system if you prefer. Both github and bitbucket provide free private git repositories for educational purposes.

You should submit your source control logs. We will be looking for small self-contained commits with clear commit messages. You can generate a git log using the command

```
git log --stat > log.txt
```

Problem set check-in

You are required to attend a short meeting with your partner and a TA. You should come to the meeting prepared to discuss your approach to the assignment, the division of labor between yourself and your partner, and any questions you may have.

Meetings will be held during normal consulting hours during the week of the 7th. You should sign up for a slot on CMS.

Part Two: Async warmup

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

Exercise 1:

Write a function `fork` that takes a deferred and two blocking functions, and runs the two functions concurrently 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 2:

Using **only** `(>>=)`, `return`, and the ordinary `List` module functions, implement a function with the following specification:

```
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` **concurrently** 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.

Update (version 2): In fact, this is **not** the same as the specification for Deferred.List.map (in fact the Jane Street documentation does not give a specification for `Deferred.List.map`).

`Deferred.List.map [1;2] f` will not begin running `f 2` until the Deferred returned by `f 1` is determined (informally, until `f 1` completes). If you want use the function specified above for your implementation, either use your implementation of `deferred_map` or supply the optional argument `~how:‘Parallel` to `Deferred.List.map`:

```
Deferred.List.map ~how:‘Parallel [1;2] f
```

Exercise 3:

Implement the asynchronous queue interface defined in `aQueue.mli`. You may find the `Async.Std.Pipe` module useful for this exercise.

Part Three: 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 folding.

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 combined — 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 1 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.

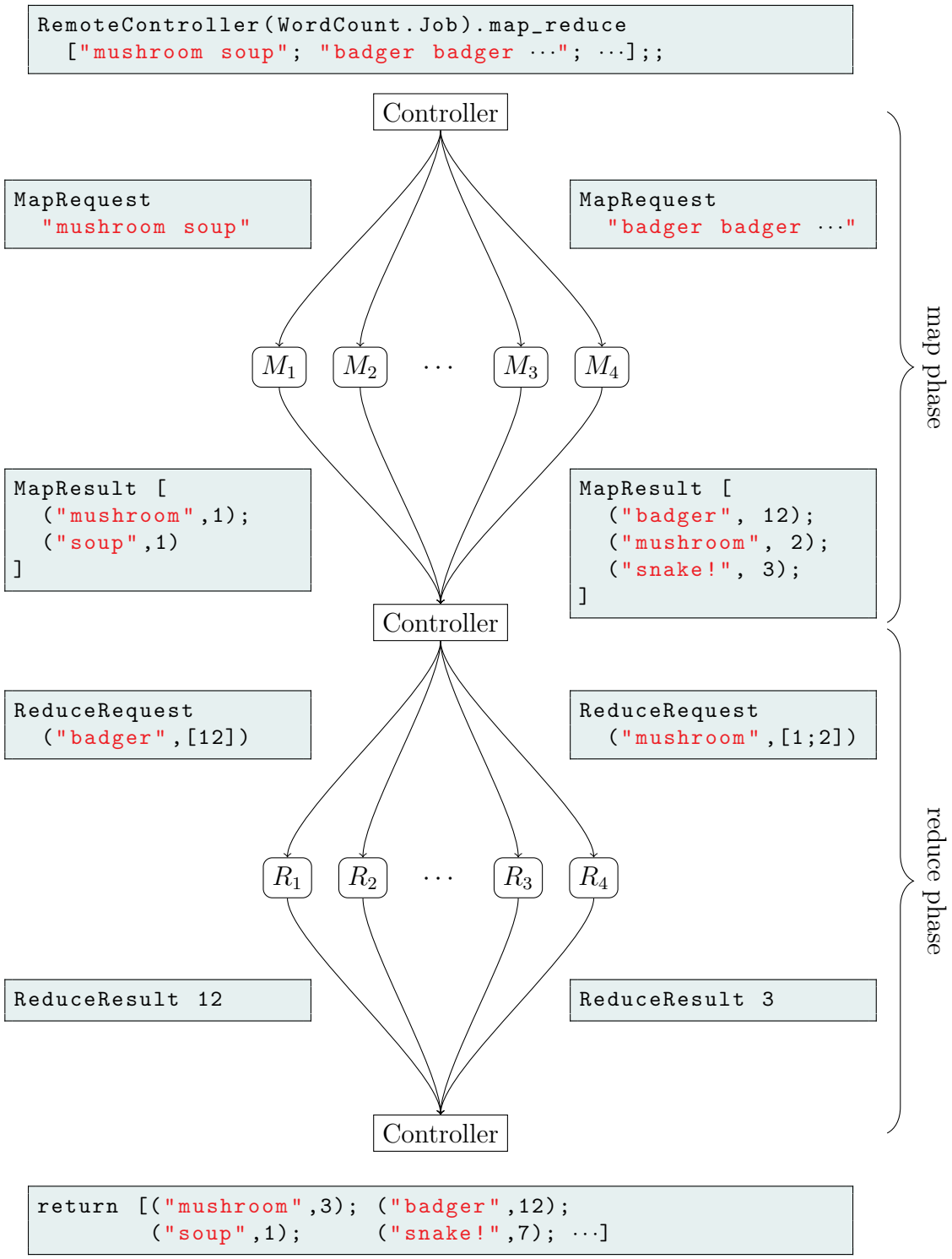

Figure 1: Execution of a WordCount MapReduce job

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.t`s and receive `WorkerResponse.t`s.

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.

> **Update (version 2):** We have added the function `Async.Std.Socket.shutdown` to our documentation, which you can call to 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 `InfrastructureFailure` 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 a `MapFailed` or `ReduceFailed` exception.

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 `MapFailed` exception.

> **Update (version 2):** We did not define the `InfrastructureFailure`, `MapFailed` or `ReduceFailed` exceptions in the release code. Feel free to define them yourself or to raise any other exception (for example by calling `failwith`.

Exercise 4: 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 <app_name> <app_args>
```

Exercise 5: 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`.

Exercise 6: [Karma] handle slow workers

It is possible that some workers may simply be slow. As an optional Karma problem, you may detect whether a worker is taking more than a few seconds to process a job, and add a second worker to process the same job if it is. If both workers are taking too long, you can add a third worker, and so on.

The old workers should not be terminated; the first of these workers to respond should determine the output of the job. You may find `Ivars` useful for this task.

Part Four: MapReduce Applications

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 wc ../writeup/ps5b.tex
```

Exercise 7: Inverted Index

An inverted index is a mapping from words to the documents in which they appear. Complete the `InvertedIndex` module (in `apps/index`) that takes in a master list of files, and computes an index on those files.

For example, if the files `master.txt`, `zar.txt` and `doz.txt` contained the following:

`master.txt`
```
zar.txt
doz.txt
```

`zar.txt`
```
ocaml is fun
fun fun fun
```

`doz.txt`
```
because fun
is a keyword
```

then running the `index` app on `master.txt` should produce the output

```
[("ocaml", ["zar.txt"]);              ("is", ["zar.txt"; "doz.txt"]);
 ("fun",   ["zar.txt"; "doz.txt"]); ("because", ["doz.txt"]);
 ("a",     ["doz.txt"]);            ("keyword", ["doz.txt"])]
```

Exercise 8: Genetic Sequence Alignment

The goal of this exercise is to identify which fragments of a DNA sequence appear within a longer reference sequence. This is an important problem from the field of computational biology.

A DNA sequence is a string made from the letters G, C, A and T. The input to your application will be a single long sequence (called the "reference") and a collection of short sequences (called "reads"). Your goal is to identify subsequences of the reads that match some part of the reference.

For example, if you are given the reference sequence

```
ref:   GATCTCTATGCAAAATACGTATTTGTACGTCCACCCTCGGAGTGGTG
```

and one of the reads is

```
read: CGTATTTGTACATCCACCCTCGG
```

your goal is to discover that they match well when lined up as follows:

```
match:                  ----------- -----------
ref:   GATCTCTATGCAAAATACGTATTTGTACGTCCACCCTCGGAGTGGTG
read:                  CGTATTTGTACATCCACCCTCGG
```

This problem can be solved using two MapReduce jobs as follows.

- In the first map phase, the input reference and reads will be broken into 10 character sequences (called "10-mers," or more generally "k-mers"). For example, the read `"AGCTAGCTCAGTACC"` would be mapped as follows:

  ```
  read X: AGCTAGCTCAGTACC
  output: AGCTAGCTCA       occurs in read X at offset 0
           GCTAGCTCAG      occurs in read X at offset 1
            CTAGCTCAGT     occurs in read X at offset 2
             TAGCTCAGTA    occurs in read X at offset 3
              AGCTCAGTAC   occurs in read X at offset 4
               GCTCAGTACC  occurs in read X at offset 5
  ```

  The mapper will output the k-mers as keys, and identifying information (such as the source sequence and offset) as values.

- The first reduce phase collects all of the occurrences of the given k-mer, and outputs a list of all possible matches between a sequence and a read.

- The second map phase will take the k-mer matches as input and will output them with keys given by the identities of the reference and read to which they correspond.

- The final reduce phase will combine adjacent shared k-mers. For example, if the 10-mers at offsets 5, 6, and 7 of a read r match the reference at offsets 17, 18, and 19, then we know that the subsequence of r of length 12 starting at offset 5 matches the reference at offset ~~15~~ 17.

At the end of these two phases, the output will be a list of partial matches between reads and references.

We have provided you with starter code to load files containing the data. You must implement the `DnaSequencing.App.run` function which takes a list of reads and references and outputs a list of matches between them.

Getting started

To help you get oriented, here is a brief summary of the files in the release:

- The `map_reduce` folder contains all of the infrastructure code
  - The `MapReduce` module defines the interface between the apps and the infrastructure.
  - `RemoteController` and `Worker` are the modules that you have to implement.
  - `LocalController` contains a working non-distributed implementation of the `Controller` interface. It makes use of `Combiner` for the combine phase.
- The `apps` folder contains the apps and the utilities that they use.
- The `async` folder contains the stubs for the async warmup exercises in part .

Comments

In addition to the usual comments, we are particularly interested in your feedback about using Async for this project.

Karma suggestions

There are a lot of fun things you can do with MapReduce. For example:

- There are a very large number of applications that can be implemented in the MapReduce framework. Find one that interests you and code it up!
- Brute force algorithms can work well when you have lots of brutes. You could adapt your solver from PS3 to run as a MapReduce app.
- You could think about how the software design would change if you wanted to make the workers themselves concurrent (and write up a summary)
- You could also think about how to extend the design so that the mappers communicate directly with the reducers to reduce the load on the controller.