CS 3110 Fall 2014 **Due at 11:59 PM, 11/13/14**
Problem Set 5
Version 1 (last modified October 24, 2014)

Objectives

- Write asynchronous programs in an event-driven style.
- Learn about the MapReduce paradigm.
- Develop a distributed system.
- Practice reading a large requirements document and producing a piece of software that satisfies those requirements.
- Practice working in a software development team.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 15 and 16; recitations 14 and 15
- Jane Street's Async documentation
- The CS 3110 style guide
- Real World OCaml, Chapter 18
- (Optional) The original MapReduce paper from Google

What we supply

We provide an archive `ps5.zip` that you should download from CMS. We also provide customized documentation of Async on the course website at `http://www.cs.cornell.edu/Courses/cs3110/2014fa/hw/5/doc/`. This documentation is designed to guide you to the most useful parts of the Async library. It's unlikely you actually want to use any part of the Async library that is not included in this customized documentation. If you spot any errors in this documentation, please let us know.

Teams and source control

You are required to work with a small team of two to four students for this problem set. Each team member is responsible for understanding all parts of the assignment. You need not use the same team members as in previous problem sets.

You are required to use `git` to work with your team. Your repository must be private. We expect the `git` log you submit to show evidence of work over a period of time, not just a single commit at the end.

Team check-in

You are required to attend a short check-in meeting with your team and a TA. You should come to the meeting prepared to discuss your approach to the problem set, especially the MapReduce problem, the division of labor among your team members, and any questions you might have. You are expected to have completed the Async warmup problem by the time you have your check-in meeting.

Check-in meetings will be held during normal consulting hours on Sunday, November 2 through Thursday, November 6. You will sign up for a meeting time on CMS; further instructions will be posted on Piazza within the next week.

Grading issues

Compilation errors: All code you submit must compile. If your submission does not compile, we will notify you immediately. You will have 48 hours after the due date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming: We use automated grading, so it is crucial that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions will be treated as compilation errors.

Code style: Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct will nonetheless be penalized. Take extra time to think and find elegant solutions.

Late submissions: Carefully review the course policy on submission and late assignments. Verify before the deadline on CMS that you have submitted the correct version.

Specification and Testing

You should continue to document specifications for functions and modules you submit as part of your solutions. And you should continue to test your code as well as possible. We will not ask you, however, to turn in your test cases.

Problem 0: CS3110 Tools Update (0 points)

Before you begin the problem set, please update your `cs3110` tools. Navigate to the release code we provide and run `update-tools.sh`.

```
bash update-tools.sh
```

Problem 1: Async warmup (30 points)

The starter code for this problem can be found in `async/warmup.ml` and `async/warmup.mli`. There are example usages in `async/examples.ml`.

Hint on teamwork: for exams, everyone is going to need to be comfortable enough with concurrent programming to solve these kinds of problems on their own. So delegating this problem to one person is not a good idea.

Exercise 1.

Write a function `fork` that takes a deferred computation and two functions, and runs the two functions concurrently when the deferred computation becomes determined, passing those functions the value determined by the original deferred:

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

The deferred values encapsulated by the two functions (i.e., the values of type `'b` and `'c`) should be ignored.

Exercise 2.

Implement the following function:

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

Ignoring concurrency, `deferred_map` should have the same input–output behavior as `List.map`. That is, both take a list `l`, a function `f`, and return the result of mapping the list through the function. But `deferred_map` should apply `f` concurrently—not sequentially—to each element of `l`.

You may use only `return`, `bind` *a.k.a.* `(>>=)`, and the OCaml standard library `List` module functions in your implementation. You specifically may not use `Deferred.List.map`, which anyway has a different type than `deferred_map`.

Exercise 3.

Have a check-in meeting, as described above. Attendance at this meeting will be worth 10 points of this problem.

Problem 2: Asynchronous Queue (20 points)

Implement the asynchronous queue data abstraction in `async/aQueue.ml`. Its interface is defined in `async/aQueue.mli`. Make sure to document the abstraction function and any representation invariants for your implementation. Your implementation should leverage the `Async.Std.Pipe` module.

Problem 3: MapReduce Framework (100 points)

Google's *MapReduce* is a framework for distributed computation that uses ideas from functional programming to parallelize applications and manipulate massive data sets in an efficient manner by distributing storage and computation across many computers, which are called *workers*. MapReduce structures computations into a *map phase*, in which workers transform independent data points, and a *reduce phase*, in which workers combine the transformed data into a result. This structure is inspired by the `map` and `fold` functions you have learned in 3110. The entire computation is coordinated by a *controller* that orchestrates the mapping, combining, and reducing.

In this problem, you will implement a distributed MapReduce framework yourself. We provide starter code in the `map_reduce` directory, which includes modules named `MapReduce`, `Worker`, and `RemoteController`, as well as other modules that we describe later.

The only files you may modify are `remoteController.ml` and `worker.ml`.

Jobs, map, and reduce. The smallest logical unit of work in a MapReduce application is a *job*, which is an abstraction that encapsulates `map` and `reduce` functions. We provide you a signature `MapReduce.Job`. It declares four abstract types—`input`, `key`, `inter`, and `output`—which we describe, next.

A job is executed as follows:

- The `map` function takes a single `input` and transforms it into a list of keys and intermediate values. The `map` function is called once for each element of the job's input list.

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

- The `map` results are then combined. Intermediate values associated with the same key are merged into a single list.

- Finally, the `reduce` function transforms a `key` and an `inter list` to the output for that key. The `reduce` function is called once per key.

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

The beauty of this design is that each call to `map` and `reduce` is independent, so the calls can be distributed across a large number of machines. This enables MapReduce applications to quickly process very large data sets.

Controllers and workers A *controller* orchestrates the execution of MapReduce. The `MapReduce.Controller` functor takes a `MapReduce.Job` and exports a `map_reduce` function, which takes in a list of `input`s and is responsible for mapping, combining, and reducing the them into a list of `key * output` pairs:

```
val map_reduce : input list -> (key * output) list Deferred.t
```

As an example, and to aid you in testing, we provide the `LocalController` module in `map_reduce/localController.ml` and `map_reduce/localController.mli`. It is a complete implementation of a `MapReduce.Controller`, but it runs only on a single, local host. It does not distribute computation across many hosts on the network; your `RemoteController`, described below, will need to implement that distribution.

Workers implement the `Worker` signature in `worker.mli`. `Worker` exports a functor that takes a `Job` and provides a `run` function, which waits for work from a controller, performs the work, and returns results using the provided `Reader.t` and `Writer.t`:

```
val run: Reader.t -> Writer.t -> unit Deferred.t
```

Distributed implementation. 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`, and responses have 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.

Files provided in the starter code:

- `controllerMain.ml` and `workerMain.ml` parse command line arguments and spawn a MapReduce controller and MapReduce worker respectively. These are the two files you will `cs3110 run`.
- `mapReduce.ml` and `mapReduce.mli` define the core types and modules you will need to implement a MapReduce controller including `Job`, `App`, and `Controller`.
- `worker.ml` and `worker.mli` define the core types and modules you will need to implement a MapReduce worker.
- `localController.ml` and `localController.mli` implement a MapReduce controller that processes all data locally without distributing it to any workers. We have implemented this controller for you.
- `remoteController.ml` and `remoteController.mli` implement a MapReduce controller that distributes work to a set of workers. You will implement this controller.
- `protocol.ml` and `protocol.mli` define the types of messages passed between the remote controller and the set of workers.
- `combiner.ml` and `combiner.mli` provides a utility functor that can combine the results of a set of workers. This functor is useful for implementing controllers.

Applications. A MapReduce *application* is a distributed computation implemented using the MapReduce framework. An 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 `Job`s. We provide the `WordCount` application, described next, as an example. You can find it in `apps/word_count`.

Example: Word Count. Figure 1 depicts a distributed execution of a word-counting application. The input to the application is a list of filenames. The word count application gathers the lines of every file. During the map phase, the application's controller sends a `MapRequest` message for each input line to a mapper. The mapper invokes the `map` function in the `WordCount.Job` module, which reads the line 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 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.

Error handling. There are two classes of errors that can arise in MapReduce:

- **Infrastructure failure.** There might be a failure of either the network or the processes implementing the MapReduce framework. Your implementation should attempt to be robust and tolerate failures of that infrastructure. For example, 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, then the controller should close the connection to the worker and continue processing the job using the remaining workers. As another example, if a worker encounters an error when communicating with the controller, it should simply close the connection.

  In the worst case, the controller may raise `RemoteController.InfrastructureFailure`. This exception would be reasonable, for example, if the controller cannot establish a connection with any workers.

- **Application failure.** There might be a failure of the application that is running on the MapReduce framework. 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 `MapFailure` or `ReduceFailure` exception. Those exceptions carry a `string`, and you may choose to supply whatever information you like in that string. For debugging, you might find it helpful to include information such as the name and stack trace of the application's exception. Those can be obtained using the `Printexc` module from the OCaml standard library.

Exercise 1: 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 `controllerMain.ml`:

```
cs3110 compile map_reduce/controllerMain.ml
cs3110 run map_reduce/controllerMain.ml <app_name> <app_args>
```

Exercise 2: 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.

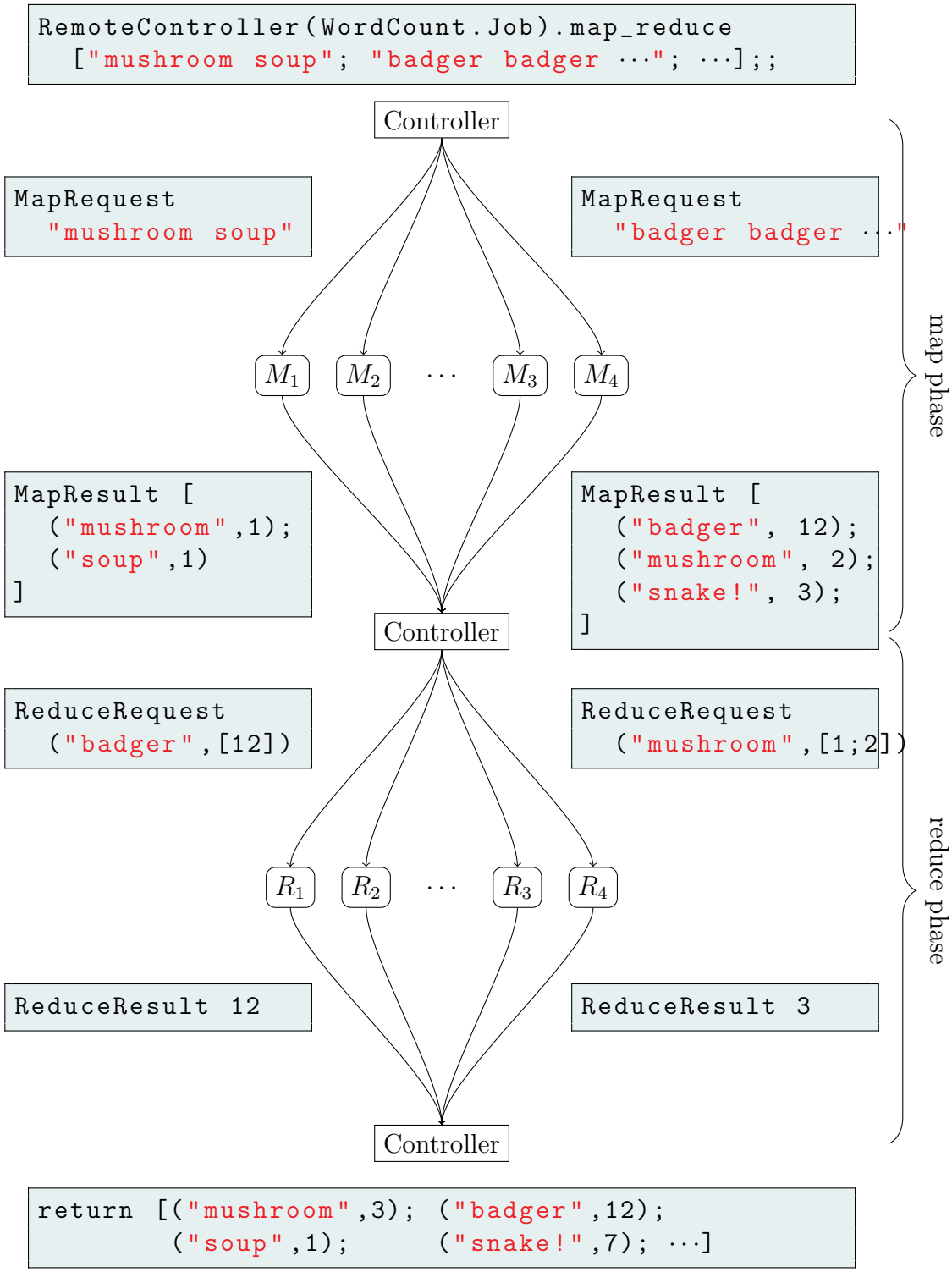


Figure 1: Execution of a WordCount MapReduce job

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

```
cs3110 compile map_reduce/workerMain.ml
cs3110 run map_reduce/workerMain.ml 31100
```

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

Problem 4: MapReduce Applications (60 points)

In this problem, you will implement a few MapReduce applications.

We have 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 map_reduce/controllerMain.ml
$ cs3110 run map_reduce/controllerMain.ml -- -local wc apps/word_count/↘
   →data/badger.txt
"soup":          1
"mushroom":      3
"snake!":        7
"badger":        12
```

Note the `--` to disambiguate the options to `cs3110 run` from the options to `controllerMain.ml`. All options that appear after the `--` will be passed to `controllerMain.ml` instead of `cs3110 run`.

Exercise 1: 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 list of files and computes an index from those files.

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

master.txt	zar.txt	doz.txt
zar.txt	ocaml is fun	because fun
doz.txt	fun fun fun	is a keyword

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

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

Exercise 2: Composition of Relations.

A binary relation R on sets A and B is a set of ordered pairs of elements in A and B. That is, it is a subset of the Cartesian product $A \times B$. If $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ are binary relations, then their composition $S \circ R$ is the relation

$$\{(x, z) \mid \exists y \in Y.\ ((x, y) \in R \wedge (y, z) \in S)\}.$$

(Recall that $\exists$ is the existential quantifier from first-order logic and means "there exists".)

Complete the `RelationComposition` module (in `apps/relation_composition`) that takes in two files and computes the composition of relations represented in the files. For example, if the files `R.txt` and `S.txt` contained

`R.txt`	`S.txt`
a, b	b, 1
a, c	b, 2
d, e	e, 3

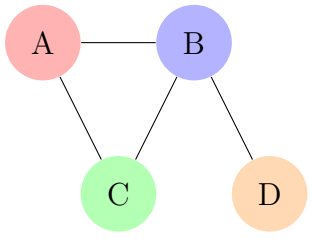
then running the `relation_composition` app on `R.txt` and `S.txt` should produce

```
(a, 1)
(a, 2)
(d, 3)
```

Exercise 3: Common Friends.

Define two vertices in an undirected graph A and B to be *friends* if there is an edge between A and B. Define a vertex C to be a *common friend of vertices A and B* if A and B are both friends with C, and furthermore A and B are themselves friends.

Complete the `CommonFriends` module (in `apps/common_friends`) that takes in a file containing the adjacency list representation of a graph, and computes the common friends for each friendship in the graph. For example, suppose that file `graph.txt` contains the adjacency list representation of the following graph:



`graph.txt`

```
A:B,C
B:A,C,D
C:A,B
D:B
```

Then running the `common_friends` app on `graph.txt` should produce

```
(A, B): C
(A, C): B
(B, C): A
(B, D):
```

Problem 5 (0 points)

[written,ungraded] In your file of written solutions, please include any comments you have about the problem set or about your solutions. This would be a good place to list any known problems with your submission that you weren't able to fix, or to give us general feedback about how to improve the problem set.

Include a statement of what work in this problem set was done by which team member. The ideal case is that each of you contributed to every problem. But (especially since this problem is ungraded) please be honest about how you divided the work.