CS 3110 Spring 2015 | **Soft deadline: 5/6/2015**
Problem Set 5 | **Hard deadline: 5/6/2015**
Version 1 (last modified April 18, 2015)

Objectives

- Write asynchronous programs in an event-driven style.
- Learn about the MapReduce paradigm.
- Develop a distributed system.
- Design a complex piece of software.
- Practice working in a software development team.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 18 - 21; recitations 18 - 20
- CS 3110's Async Documentation
- Jane Street's Async documentation
- The CS 3110 style guide
- Real World OCaml, Chapter 18
- (Optional) The original MapReduce paper from Google

Grading issues

Names and types: You are required to adhere to the names and types given in the problem set instructions. If your code does not compile against and pass all the tests in the provided public test file(s), your solution will receive minimal credit.

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.

Part 1: Project overview (20 points)

In this problem set you will be implementing a simplified version of the MapReduce framework, and a handful of applications that use your framework.

Unlike previous problem sets, we are giving you a very small amount of starter code. You will be responsible for both the design and implementation of your software. Therefore it is imperative that you carefully document your design, and that you communicate well with your team.

Exercise 1: 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.

We strongly recommend that you use `git` to manage your source code. Be sure that your repository is private!

Exercise 2: Design review.

You are required to attend a short design review meeting with your team and a TA. You should come to the meeting prepared to discuss your approach to the problem set:

- How will you divide the software into modules? What will be the interfaces between those modules?
- What will a developer have to do to add new apps to the system?
- How will you share the work between group members?
- Are you factoring out common code, for example between the local and remote implementations?
- How will you manage concurrent access to the workers?
- What types of messages will you send over the network?
- What kinds of errors might occur at runtime? How will you discover them and handle them?

The design review is also a great opportunity to discuss any questions you have with a TA.

Design reviews will be held during normal consulting hours starting on Thursday, 4/23. You will sign up for a meeting time on CMS; further instructions will be posted on Piazza.

Exercise 3: Documentation.

Because you are responsible for most of the design, it is important that you tell us about your submission. Your documentation should include the following:

- Documentation for **users:** instructions on how to run your software. Be sure to mention any known bugs or extra features.
- Documentation for your **elective app:** what it does and how it works.
- Documentation for **app makers:** useful information and instructions on how new apps can be added to the system
- Documentation for **developers:** an overview of the structure of your code. Information that would be useful to someone who wanted to debug or modify your implementation. Information about tests and specifications.
- Documentation for **course staff:** anything else you want to tell us about the problem set or your implementation. Also let us know how you divided the work between partners.

Your documentation should be contained in a file called `README.txt` or `README.pdf`.

Part 2: MapReduce Framework (100 points)

Google's *MapReduce* is a framework that uses ideas from functional programming (namely mapping and folding) to distribute large computations over many networked computers. The computers that do the computation are called *workers*; there is also a single *controller* that is responsible for sending work to the workers and collecting the results.

A MapReduce application runs in three phases. In the first phase (called the *map* phase), the workers independently transform each input datum into a collection of (key, value) pairs. In the *combine* phase, all of the intermediate values with the same key are collected together. In the final *reduce* phase, the workers independently transform the collection of data corresponding to each key to produce an output value. Finally the controller collects these final values and outputs them.

This framework has a lot of flexibility, because each application can have its own functions for mapping and reducing the data (as you know, a very large number of functions can be written using `map` and `fold`!). The framework also enables scalability, because each call to `map` and `reduce` is independent; the computation can be distributed across a large number of machines.

Example: Word Count. Figure 1 depicts a distributed execution of a word-counting application. The input to the application is a list of lines.

During the map phase, the application's controller sends each input line to a mapper. The mappers split the lines up into words; for each word, the mapper outputs the word as a key, and the number of times it occurs as a value. The mapper then sends these pairs back to the controller.

Once the controller has collected all of the intermediate `(word,count)` pairs, it executes the combine phase. During the combine phase, all of the pairs having the same word are collected into a single list.

Finally, the controller begins the reduce phase. In the reduce phase, the controller sends each (word, list of counts) pair to a reducer. The reducer sums up all of the counts, and returns the pair (word, total) back to the controller. The controller then outputs the entire list of words and totals.

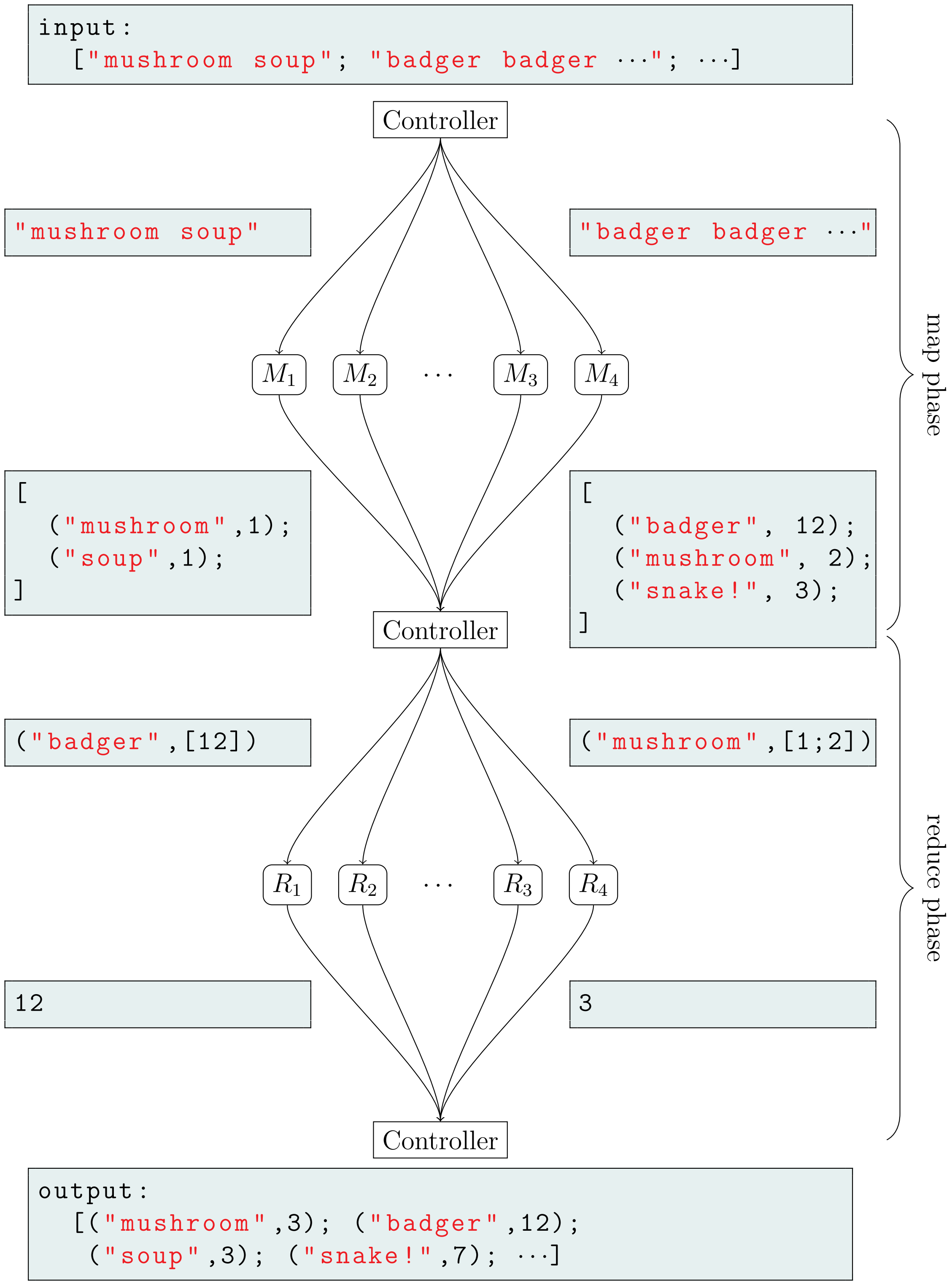

input:
["mushroom soup"; "badger badger ⋯"; ⋯]
Controller
"mushroom soup"
"badger badger ⋯"
M_1
M_2
⋯
M_3
M_4
map phase
[("mushroom",1); ("soup",1);]
[("badger", 12); ("mushroom", 2); ("snake!", 3);]
Controller
("badger",[12])
("mushroom",[1;2])
R_1
R_2
⋯
R_3
R_4
reduce phase
12
3
Controller
output:
[("mushroom",3); ("badger",12);
("soup",3); ("snake!",7); ⋯]

Figure 1: Execution of a WordCount MapReduce job

Your MapReduce tasks

Exercise 4: Implement local MapReduce.

Although MapReduce applications are intended to be run over a network, it will be helpful for testing if you are able to simulate your apps within a single process.

In the file controllerMain.ml, implement the function run_local. When you run

```
cs3110 run controllerMain.ml -- -local app_name arg1 arg2 ...
```

from the command line[1], our starter code will call

```
run_local "app_name" ["arg1"; "arg2"; ...]
```

When the deferred returned by run_local becomes determined, the program will terminate. run_local should execute the map, combine, and reduce phases on the given application.

Exercise 5: Implement distributed MapReduce.

To implement the distributed version of MapReduce, you must implement the functions ControllerMain.run_remote and WorkerMain.serve.

ControllerMain.run_remote will be called when you run controllerMain.ml without specifying the -local option. It should call Protocol.connect_to_workers which will establish connections to all of the workers listed in addresses.txt. run_remote should then distribute the input to the workers to perform the map phase. The workers will return the output of the map phase to the controller. The controller should combine the intermediate results, and then distribute the keys and combined values to the workers. The workers should perform the reduce phase, sending the output back to the controller. Finally the controller should display the output. When all of this is done, the deferred returned by run_remote should become determined, and the system will exit.

WorkerMain.serve will be called when a controller establishes a connection to the worker (by calling Protocol.connect_to_workers). It should respond to all of the requests that the controller sends it.

The Protocol module contains a useful functor for sending typed messages across the network. Protocol.Make takes a module containing a type t and provides a send and a receive function for communicating values of type t. See the echo directory for an example using the Protocol module.

Take care when using the Protocol module that the types of the messages you are sending match the types of the messages you are receiving. Because the Protocol module circumvents the type system, receiving a message of the wrong type can cause your program to fail in arbitrarily surprising ways.

[1] 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.

Part 3: MapReduce Applications (50 points)

Exercise 6: Word count.

Implement the word count application described above. To run the word count application, a user should execute

```
cs3110 run controllerMain.ml word_count <filename>
```

The word count application should call `WordCountUtil.read <filename>` to read in the corresponding file, should run the word count application, and should then call `WordCountUtil.show` to display the output.

The `apps/word_count` directory contains the `WordCountUtil` module and some sample data.

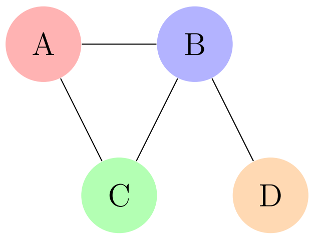
Exercise 7: Common Friends.

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.

Implement an application called `"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.

You can make use of the `read` and `show` functions in the `CommonFriendsUtil` to read an adjacency list from a file and to output the resulting list of common friends.

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

```
cs3110 run controllerMain.ml common_friends graph.txt
```

should produce the output

```
A B C
A C B
B C A
B D
```

Exercise 8: Elective app and tournament.

You are responsible for implementing a third application of your own choosing. This app should make effective use of the MapReduce framework to distribute its work across many workers.

If you are feeling uninspired, we have provided a description of many of the applications from previous semesters, including a bitcoin transaction processor, a DNA sequencing algorithm, and a search engine. However, we strongly encourage you to go wild!

At the end of the semester, we will have an (optional) demo session to give you a chance to show off your creations to your classmates and the staff. A panel of judges will choose their favorite app for induction into the CS3110 hall of fame. This is your chance to acheive CS3110 immortality!