Homework #6: Building a Distributed Map/Reduce Framework

Due Thursday, May 2nd at 11:59 p.m.

In this assignment you will implement a simple map/reduce framework. Map/reduce is a programming model for processing large data sets, typically implemented by distributing a computation on a cluster of computers where the data is stored. The computation, provided as a plug-in to the framework, is specified as a separate *map* task and a *reduce* task. These tasks are typically submitted to a master server which manages the computation until it is complete, providing robust computation even if some worker servers fail or are unavailable during the computation.

Your goals in this assignment are to:

- Gain a deep understanding of map/reduce and the challenges of building a fault-tolerant distributed system by implementing a map/reduce framework.
- Learn how to use a map/reduce framework by implementing a simple map/reduce task that uses your framework.
- Practice network socket programming within Java.
- Practice parallel and concurrent programming within Java.

For this assignment we encourage you to discuss map/reduce and high-level details of your solution with the course staff and with your classmates, but you may not share code for this assignment or see others' code, and you must submit your own solution for this assignment.

This document continues by describing the requirements of your map/reduce framework and the map/reduce computation you must implement to help test your framework.

Design and implement a map/reduce framework

Your framework must match the basic master/worker architecture as described in lecture. Specifically, your framework should allow a map/reduce task to be submitted to a master server, which then manages the task until it is complete. Worker servers in your system should perform the actual work of the map/reduce task, executing the map and reduce tasks on subsets of the data they locally store or can retrieve.

The master server first distributes the map task among worker servers in the map/reduce system, with each worker executing the map task on the subset of the data it stores. After the map task is complete, the master server distributes the reduce task among worker servers – sometimes but not necessarily the same workers that executed the map task. Each reduce worker obtains a subset of the results of the map task, a step known as *the shuffle* because data is seemingly shuffled between map and reduce workers. The reduce worker then sorts its subset of results and executes the reduce task on its subset. When the reduce task is complete, the master server finally obtains and returns the result of the reduce task to whomever submitted the computation.

If a worker server crashes (or otherwise fails to return a result) during the computation, the master server simply distributes the map or reduce task to another worker, which either executes the map task on a replica of the data subset or obtains the results of the map task and executes the reduce task on the appropriate subset of the data.

A faux distributed storage system

To achieve efficient operation, map/reduce is typically coupled closely to an implementation of a distributed storage system. In this assignment, however, we have not provided a distributed storage system and you should **not** implement your own.

Instead, we have provided sample data partitioned much as it might be partitioned within a distributed storage system, into separate *tablets* that could each be replicated and served from multiple storage servers. You may copy all of this data to any server you use to test your solution, but your worker program should mimic the behavior of a server within the storage system by only accessing a limited subset of the data.

For this assignment our expectation is that each tablet is locally available at one or more worker servers. You may assume that the worker-to-tablet association is fixed and known by all workers and the master program. When your master program distributes a map task among worker servers, it should assign each map worker to execute the task on a subset of the tablets that worker is purportedly hosting.

For testing purposes, you may configure your worker-to-tablet association in any reasonable way. You may, for example, read the configuration from a configuration file or hard-code the association directly into your programs. For example, in the `assets/tablets.cfg` file we list a fixed worker-to-tablet association that mimics a three-server storage system where each tablet is replicated onto two servers. With this configuration, our expectation is that the map/reduce system should compute a correct result with any single worker failure, but that two worker failures would make some tablet data unavailable and thus prevent the map/reduce implementation from correctly completing a computation on the full set of tablets. (If some tablet data is unavailable, any behavior of the map/reduce system is OK.)

To simplify your work we have provided a `Tablet` class that, given a tablet name (e.g., `"7"`), allows you to iterate over all files stored in that tablet. This `Tablet` class assumes the files are available on the local file system. Our sample data – books from Project Gutenberg – are partitioned into ten separate tablets.

Our `MapTask`, `ReduceTask`, and `Emitter` interfaces

We have provided a plug-in interface for computations for your map/reduce framework, as well as a sample `WordCount` computation you may use to test your implementation. Like the map/reduce example from class, our plug-in interface requires the results of both map and reduce tasks to be `String/String` key/value pairs. Your framework is not required to use our plug-in interface, but we recommend that you use our interface unless you have a compelling reason to change it.

Based on the example map/reduce computation from class, our `MapTask` and `ReduceTask` interfaces should be mostly self-explanatory. These interfaces use an `Emitter` interface that allows `MapTask` and `ReduceTask` implementations to communicate their results – each result being a single key/value pair – to the framework, which may then process those results as needed. We also provide a simple `Emitter` implementation, `OutputStreamEmitter`, that emits results to an arbitrary `OutputStream`. If you use our `OutputStreamEmitter` you should probably implement a class that allows you to read its emitted key/value pairs from an arbitrary `InputStream`.

The robustness of your solution

Because one of the most difficult aspects of building a distributed system is making the system robust to failure, you could potentially make your solution arbitrarily complex by defending against complex partial failures. Our goal, however, is for your solution to be somewhat robust to simple failures without robustness becoming a substantial burden.

Specifically, at minimum your solution should satisfy the following requirement: If worker servers do not crash during a computation, your solution should correctly complete the computation if all the tablets are available on some worker, even if some worker servers are unavailable. You do not need to be robust to arbitrary partial failures or failures of the master server. You can evaluate this requirement through the following procedure: (1) configure your system to use multiple servers, (2) don't start the worker server program on some of those servers even though the system is configured to use them, and (3) configure the tablet-to-worker associations so that all tablets are available on some worker where the program is running. With this procedure your master server might attempt to contact unavailable workers, but all data needed for the computation should eventually be obtained and the computation should be correctly completed.

If you want to challenge yourself you may attempt to make your system more robust to more complex failures, including failures of worker systems in the midst of a computation or failures of the master program. You can simulate such failures by arbitrarily killing your worker Java programs during a computation, or even write Java code within your program to simulate such a failure. (E.g., `if (random.nextDouble() < 0.001) System.exit(42);` might help.)

Using your map/reduce framework

When you have completed your map/reduce implementation, you must demonstrate how to use your map/reduce framework by executing our sample computation on your framework and then implementing a new map/reduce computation of your own. We describe these two tasks below.

A sample `WordCount` computation

We have provided a sample map/reduce computation to count all occurrences of all words in a corpus of data, as `WordCountMapTask` and `WordCountReduceTask`. This computation was also provided as an example in lecture.

In the `tasks` package, create a `WordCount.java` program that executes our computation on your framework. You may adapt our implementation as necessary if you changed our plug-in interface. If your framework is well-designed and also uses our `MapTask` and `ReduceTask` interfaces, you should be able to submit this computation as a map/reduce task with just a few lines of code.

You may also use this computation to test your map/reduce framework, but we recommend you configure your framework to use just a small data set until you believe that your framework is correct. For reference, the word "a" appears 9976 times in the small data set we provided within your SVN repository, and 149179 times in the larger data set we provided on Piazza.

Suggesting words based on word prefixes

When you have tested your solution using our `WordCount` example, write a map/reduce task to help determine the best word completions for word prefixes. Specifically, your map/reduce task should analyze a corpus of data and output a *prefix/word* key/value pair for each prefix that appears in the corpus, where the word output for each prefix is the word most likely to complete that prefix within the corpus.

For example, many words start with "a" and thus the prefix "a" could complete many words within a typical corpus. For most corpuses, however, the most common word beginning with "a" is the word "a" itself, so the output of your map/reduce task will probably include the key/value pair `a/a`. Similarly, the prefix "altru" could start one of several words – altruism, altruist, altruistic, etc. In our larger sample corpus, however, "altruist" appears twice and each of the other "altru" words appears only once, so the output should include the key/value pair `altru/altruist` if run on that corpus.

Although it is possible to implement this map/reduce task efficiently, it is likely that your implementation will be somewhat inefficient when run on a simple map/reduce framework. We recommend that you do not try executing this task on the larger sample data set we provide via Piazza.

Evaluation

As with much of this course's work, the requirements of this assignment are intentionally underspecified to allow you substantial flexibility in the design and implementation of your solution. Your solution, however, should meet the following minimal requirements:

- Your solution should include at least three Java programs: one program for the worker servers and one program for each task (`WordCount` and `WordPrefix`) to be executed at the master server. The worker and master programs should communicate using network sockets. Your solution may use more than three Java programs if you want.
- Map workers should obtain data directly from the file system but only access "local" data in the faux distributed storage system. Reduce workers may save results directly in the file system or return the results over the network to the master server.
- You must use network sockets to implement the map/reduce shuffle, transmitting map results over the network from map workers to reduce workers. Map workers may temporarily save map results in the file system if you want, but your reduce workers should not directly access those results via the file system.
- Your solution should be robust to unavailable workers as described in the Robustness section above.
- You must include a `MapTask` interface and a `ReduceTask` interface. We recommend that you use our interfaces unless you have a compelling reason to modify them. If you modify our interfaces you must also modify our `WordCount` implementation to match your modified interface.
- Include a `README.txt` or other similar file describing how the course staff should run your framework on multiple computers. When testing your solution the course staff will, at the very least, attempt to execute the `WordCount` task on a two-server system.

Overall, this homework is worth 100 points, plus you may earn 10 points extra credit. We will grade your work as follows:

- Working map/reduce framework using separate master/worker programs and network communication when deployed on multiple servers: 40 points.
- Robustness of map/reduce framework when some worker servers are unavailable but tablet data is available: 20 points.
- Creation of the `WordCount` program that executes our map/reduce example task on your framework: 10 points.
- Correct implementation of the word-suggestion computation for your framework: 20 points.
- Javadocs and style: 10 points.
- Extra credit: Complete this assignment such that FindBugs reports no errors or warnings at the standard warning level for your solution: up to 10 points.