Homework #6: Building a Distributed Map/Reduce Framework

Due Thursday, December 5 at 11:59 p.m.

In this assignment you will implement a distributed map/reduce framework. Map/reduce is a programming model for processing large data sets, typically implemented by distributing a computation across multiple worker servers where the data is stored. The computation is provided as a plug-in to the framework, and is specified as a separate map task and a reduce task. A client submits the two tasks to a master server, which is in charge of distributing the tasks across the worker servers in the system and returning the result back to the client when it is complete.

Your goals in this assignment are to:

- Gain a deep understanding of map/reduce and the challenges of building a 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 programming using `Socket`s.
- Practice parallel and concurrent programming in 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 your code with other students and must submit your own solution for the assignment.

Designing and implementing your map/reduce framework

Your framework must match the basic architecture described in lecture. Specifically, your framework should consist of three main components: a client, a master server, and multiple worker servers. The client simply submits the map and reduce tasks to the master server and waits for the master server to confirm that the computation is complete. Upon receiving the map and reduce tasks from the client, the master server manages the map/reduce computation, distributing the map and reduce tasks across a set of worker servers and specifying which worker servers should execute those tasks on which subsets of the data. The worker servers perform the actual work of the computation, each executing the map and reduce tasks on some subset of the data.

A map/reduce computation consists of two phases: a map phase and a reduce phase. In the map phase the master server distributes the map task across a set of worker servers.

The master server specifies on which subset of the data each map worker should execute the map task, and the map workers execute the map task on that subset of the data and write *intermediate results* to a file on their local disk.

After all map workers have completed the map task the master server begins the reduce phase, distributing the reduce task across worker servers—sometimes but not necessarily the same set of workers that executed the map task. The master server specifies on which subset of the intermediate results each reduce worker should execute the reduce task. Each reduce worker begins by obtaining the specified subset of intermediate results from the map workers, a step known as *the shuffle*. In the shuffle it's often necessary for every reduce worker to obtain some intermediate results from every map worker; in practice implementing the shuffle efficiently is one of the most complex aspects of a map/reduce framework. At the end of the shuffle, each reduce worker has a disjoint subset of the intermediate results. Each reduce worker then executes the reduce task and saves the final key/value pair results in its file system, notifying the master server when it is complete. When the master server is notified that all reduce workers are complete, the master server notifies the client that the computation is complete and specifies the file locations that store the final results.

If a worker server crashes or otherwise fails to return a result, the master server distributes the map or reduce task to another worker. In a typical map/reduce framework the data is redundantly replicated in a distributed file system, and the map/reduce framework should produce the correct result any time the full data set is available on non-failed servers.

A faux distributed storage system

To achieve efficient operation, map/reduce is typically coupled closely to an implementation of a distributed storage system, which stores both the source data and the results of the map/reduce computation. For map/reduce to be efficient the map/reduce worker servers are usually servers in the distributed storage system; each map worker usually just computes the map task for a subset of the data it locally stores for the distributed storage system, and each reduce worker stores the map/reduce results in the distributed storage system, but in files stored at the reduce worker's local file 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 file sets that could each be replicated and served from multiple storage servers. For testing purposes, you may copy all of this data to any server you use to test your solution, but your worker server should mimic the behavior of a server within the storage system and only access the subset of the data specified as available to that server.

For this assignment our expectation is that each partition is locally available at one or more worker servers. You may assume that the worker-to-partition association is static and known by all workers and the master server when those programs start. When your master server distributes a map task among worker servers, it should assign each map worker to execute the task on a subset of the data that worker is hosting.

To simplify your work we have provided a `Partition` class that, given a partition name (e.g., `"7"`) and worker name, allows you to iterate over all files stored in that partition for that worker. This `Partition` class assumes the files are available in a directory layout specific to our faux distributed storage system, and we have provided sample partitioned data in your Eclipse project.

The `MapTask`, `ReduceTask`, and `Emitter` interfaces

We have also provided a plug-in interface for computations for your map/reduce framework, as well as a sample word count 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.

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.

Completing your map/reduce framework

To complete your map/reduce framework you must write the `AbstractClient#execute()` method, the `MasterServer#run()` method, and the `WorkerServer#run()` method, as well as design and implement any other components (such as implementing the `Emitter` interface) you need for `MapTask` and `ReduceTask` implementations to obtain data from and communicate results to the framework.

When you are done, you should be able to run the `MasterServer`, `WorkerServer`, and a client (such as the `WordCountClient`) as separate Java programs. The client should submit a map task and reduce task to the master server, which should manage the map/reduce computation as described above. The map/reduce computation should be executed on all files (excluding replicas) in the faux distributed storage system. All communication between the client and the master server, between the master server and worker servers, and between pairs of worker servers should use network sockets. (To be clear: your framework must use network sockets to transmit intermediate results from map workers to reduce workers during the shuffle, even if those workers are being simulated as threads in a single Java program.) The master server should assign tasks to workers in such a way that (1) each worker reads and writes only data available to that worker in the faux distributed storage system, and (2) the map/reduce computation obtains the correct result any time the full data set is available, even if some worker becomes unavailable during the computation.

Our `MasterServer`, `WorkerServer`, and client implementations contain `main` methods that start separate master, worker, and client programs. When run without any command-line arguments these programs obtain a sample master and worker configuration from two Java properties files, `master.properties` and `worker.properties`, initializing master, worker, and client classes with hostnames and ports that you can use to create network sockets for inter-process communication. By default, the `WorkerServer` simulates multiple workers in a configuration by running multiple threads in a single Java program.

Inside the `worker.properties` file we've defined a sample configuration of four worker servers, with each worker server storing a subset of the data in the faux distributed storage system and each partition being stored redundantly by at least two workers. In this configuration your map/reduce framework should be able to successfully complete a map/reduce computation if any single worker server fails during the computation.

Alternatively, the configuration for the `MasterServer`, `WorkerServer`, and client programs may be specified as command line arguments. You may specify a configuration on the command line to enable the worker servers to run as separate processes (rather than as separate threads within a single process) to facilitate testing with worker failures.

Using your map/reduce framework

When you have completed your map/reduce implementation, you must 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 tasks below.

A sample word count computation

We have provided a sample map/reduce computation to count all occurrences of all words in a corpus of data, as a `WordCountMapTask`, a `WordCountReduceTask` and a `WordCountClient` that reads our master and worker configuration files. This computation was also provided as an example in lecture. You may use this computation to test your map/reduce framework. For reference, the word "a" appears 9976 times and "and" appears 16299 times in the sample data set we provide in your Git repository.

Suggesting words based on word prefixes

After you have tested your solution using our `WordCountClient` 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 our corpus, however, the most common word beginning with "a" is the word "and", so the output of your map/reduce task should include the key/value pair `a/and`. 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.

Evaluation

Your solution should meet the following minimal requirements:

- Map workers should write their intermediate results directly to the file system. Reduce workers may return the final key/value pairs over the network back to the master server, or may write the final key/value pairs to the file system and return (to the master server) the names of the files storing the result data.
- You must use network sockets to implement all communication between the client and master, between the master and each worker, and between pairs of worker servers. Workers must use network sockets to implement the shuffle (even if the intermediate results are available in files in the test configuration).
- Your framework should be designed to work with an arbitrary (but fixed) number of worker servers. For example, if we add a worker server to our `workers.properties` configuration file, your system should correctly implement map/reduce including that worker server, without requiring any programmatic changes to your solution.
- Your framework should be robust to simple worker failures, and should compute the correct result as long as the full set of data is available on non-failed worker servers. Your framework is allowed to compute an incorrect result for a map/reduce computation if enough workers fail to make some partition data unavailable.

Overall, this homework is worth 90 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: 50 points.

- Robustness of map/reduce framework when some worker servers are unavailable but the full data set is available: 10 points.
- Correct implementation of the word-prefix 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 (15) for your solution: up to 10 points.