CSCI 5105 Programming Assignment 3: A Simple Map/Reduce-like Compute Framework

Mark Wagy (3061917, wagyx001)

April 14, 2012

1 Design Document: System Design

In this document, we describe a system that performs a simple Map/Reduce-type sort operation on files of data. The system takes a file for sorting and outputs the sorted version of that file. The sort operation is performed on a series of distributed nodes that can exist on any network-accessible machine. The result is a system that is able to sort large files quickly, and one that is robust to faults in the nodes. Various fault tolerance devices ensure that the sort operation completed successfully even if some of the nodes are unable to complete their task. The system consists of three main components: The server, client and nodes.

The system is written in the Java programming language. The communication between Server, Client and nodes is implemented using Java Sockets and the Server and Nodes extend the Java Thread class. At first, Java's Remote Method Invocation (RMI) was used, but a problem was encountered: when the Node starts up, it contacts the Server to tell the server what its contact info is (IP address and port number). This results in a *Catch 22*-type situation, because in order for the Server to be started, it needs to know where the RMI registry of each Node is running, but the Nodes need the same info for the Server in order to tell the Server where it is running. The Node needs the server to be running to get started, but the Server also needed the Nodes to be running before starting. As such, the system was rewritten using Sockets.

The actual communication taking place between elements of the system (Client, Server and Nodes) relies on a *Protocol* class, which mainly maintains a list of different types of operations that are being requested to be performed. This *Protocol* class is *Serializable* so that it can be passed as the first object in each Socket communication. Using the *Protocol* type at each Socket connection dictates, for each component, what the sequence of operations is to be undertaken so that the communication between elements can happen successfully. The *Protocol* type is used to *dispatch* an operation (and a method called *dispatch()* can be seen in both the *Node* and *Server* classes doing just that - no such method is needed for the *Client* class since it is not requested to perform operations but just does request*ing*).

1.1 Server

The server, made up of the *Server* class, is the coordinator of activities in the system. Also, it is the point of contact for the Client - the user interface for interacting with the Map Reduce system. The Server takes a request from the Client for a sort operation and maps the task to a list of available nodes. Each node gets a chunk of the sorting work that have been *Mapped* by the Server and then each node proceeds to sort its own mapped chunk independent of other nodes.The server then takes each of the chunks of sorted data that is given it by each of the nodes and randomly chooses a node to perform the Reduce task. The Reduce tasks consists of merging together each of the sorted chunks of data into a single, sorted list of rows. This merging tasks is more efficient that having to do a sort on the entire file of data. At this point the Server returns the sorted result to the Client. The Server also maintains statististics about each of the nodes and maintains and computes aggregate statistics such as the total number of requests that have been given it so far and the average load that the nodes have had throughout sort operations.

1.2 Client

The Client (the *Client* class) is the point of interaction with the user. After starting the Client, the user is requested to choose which file is to be sorted. The Client then contacts the Server to send the requested file for processing. After the file has been sorted, the Server returns a sorted version of the file. At this stage, the Client asks the user where the sorted file should be placed and the Client writes the sorted version of the file to that location.

1.3 Node

The nodes (each instances of the *Node* class) are the true workhorses of the system. They do all of the sorting tasks that are asked of them by the server. When they start up, they also report their location to the server so that the server can maintain something of a "task manager" for sort operations. When a node goes down, the list of nodes that are available are updated with that information and another node is requested to do the sorting tasks that has yet to be completed so that the system is resilient to node failures. Additionally, the nodes each check their load to assure that they have enough resources (as defined by a load threshold property in the *node.properties* file) and if they are too busy to sort, the work is handed off to another node that does have the available resources. Per the requirements, if there is no node that has the available resources to sort, then the task comes back to the original sort node to complete. Each node outputs an intermediate set of results (per the requirements) when it sorts its chunk of data into a file called *intermediate.sorted*. A mechanism has been implimented to cause nodes to voluntarily fail (also per the requirements), which demonstrates the system's robustness to node failures. This mechanism is a comparison of a value - defined in the *node.properties* file as *FailureProbability* - to a randomly generated number during the sort operation to determine if the node will be forced to fail.

1.4 Data

Additionally, classes used to represent the data were implemented: *DataPackage* and *DataRow*. These were simple abstractions used to house the data and simplify the organization and writing of the data as it was being passed between each communicating element of the system.

2 User Document: Running the System

In order to run the system some assumptions are made regarding the tools available on the user's computer: a recent installation of Java (at least Java 1.5), an installation of Ant and a Bash shell is needed to run the startup scripts.

To build the system, first unzip the package containing the system. Then change directories to the 'main' project directory (*pa3/*). Once in *pa3/*, type:

```
>> ant clean
>> ant jar
```

This will build the system and it will be ready for the user to run the startup scripts for the different system components. The user should go through this same process for each machine that they wish to run a component on (i.e. copy the system package zipped file onto the machine to run on and unzip it, then build with above Ant scripts).

The first process to be started is the server. After that nodes can be started and then a client should be run in order for the user to submit a file to be sorted.

To start the server, simply use the provided bash-script startup script called *server.sh*, which is in the main project directory. It does not take any arguments and can be invoked simply by typing:

```
./server.sh
```

Once the server has been started, it will tell the user where it is running (its IP address and port number). The user should use these values as (space-separated) arguments for the other programs: the client and nodes.

To start the node, use the *node.sh* script provided in the main project directory. It takes arguments of the server's IP address and the server's port number. So, for example, if the server resides at 128.11.22.111:9876, then the user should use the following command:

```
./node.sh 128.11.22.111 9876
```

Similarly, the client has the same arguments for its invocation (server IP and port number) and can be invoked using the *client.sh* command as follows (with the same server information as in the node example):

```
./client.sh 128.11.22.111 9876
```

The client will ask for a file to sort and then for a place to stored the sorted file.

A file in the main project directory called *node.properties* can be altered to control the frequency of (forced) failures in the nodes and the load that a node is able to handle before trying to pass work off to others. The variable to control the failure probability is *FailureProbability* and the variable that controls the load amount that a node can handle is called *LoadThreshold*.

3 Testing Description

A series of tests were run on the system, including on the same machine and will each component running on different machines. It was verified that the system works well in both the case of running on the same machine and when running on multiple machines. The majority of testing was done on multiple machines, using University of Minnesota Linux machines (kh4240-01,kh4240-02,kh4240-03,kh4240-04,kh4240-05,kh4240-06,kh4240-07, and kh4240-08).

Files of random integer values were generated for the test files. These files were sent through the system for testing. Files of different lengths were used (between one thousand and one million rows of random numbers) to verify that the system was able to handle multiple lengths of file and throughout varying load scenarios. These test files can be found in the *test* directory of the project, named with the number of rows of random integers that are in the file (for example *test/test1000k.txt* for a the test file with a million rows). To run these tests, one must only refer to the appropriate test file when asked for which file to sort when using the client.

Different load thresholds and failure probabilities were used in testing and with different numbers of nodes operational. The maximum number of nodes that were tried was seven nodes running at one time.

An experiment was performed to track the performance of the distributed sorting system using different numbers of nodes. I started with the hypothesis that a system containing more nodes would lead to better performance. The results are summarized in the following table:

Number of rows in sort file: 1 million. Failure Probability: 0.5 Load Threshold: 0.7

Number of Nodes	Time to Sort (ms)
1	33422
2	33596
3	39909
4	32949
5	30043
6	37728
7	33266

Table 1: Comparison of node number to time it takes to sort 1 million rows

The results are not very intuitive and do not conform to the initial hypothesis. But it is very likely that there are many factors at work. One such possibility is that node failures could increase with more nodes. If node failures have a larger impact on the sort time, this might lead to longer sort times - i.e. it might take significant time to find other nodes. In order to try out this hypothesis, I changed the failure probability to a very small number (0.1) and the load threshold to a very large number (10.7) to see if a shorter sort time would result (using six nodes). The result was that the sort time was 37348 milliseconds. This is a smaller number than with higher failure probability and lower load threshold as fit with my intuition, but to really be sure a larger such study would need to be undertaken. Also it might take a longer time to read/write from sockets than the actual sort operation. Having more nodes might mean longer socket IO times, thus having an impact on the timing.

Various statistics are reported including load thresholds, number of requests, average load amongst the nodes, and the time that it took to sort the files. The client reports the time it took, and the nodes themselves and the server all report some of these statistics.