CSCI 5105: Introduction to Distributed Systems

Spring 2016

Instructor: Abhishek Chandra

Programming Assignment 3: A simple MapReduce-like compute framework

(*Due: Apr/20/2016 – 11:59pm)*

1. Overview

In this programming assignment, you will implement a simple MapReduce-like compute framework for implementing a sort program. The framework will receive jobs from a client, split job into multiple tasks, and assign tasks to compute nodes which perform the computations. This framework would be fault-tolerant and should be able to guarantee the correct result even in the presence of compute node faults.

2. Project Details

In this project, you will implement 3 components.

1. **Client**: The client sends a job to the Server. The job should include a filename which stores data to be sorted. When the job is done, the client will get the filename as a result of job which stores sorted data.
2. **Server**: The server receives the sorting jobs submitted by the client. It splits the job (input data) into multiple tasks (chunks) and assigns each task to a compute node. The server needs to print the status of job e.g., how many tasks are running, how many tasks are complete for a job and etc. When the job is completed, the server prints the elapsed time to run it along with the output filename which includes sorted output returned back to the client.
3. **Compute Nodes**: Compute nodes will execute tasks (either sort or merge) sent to them by the server. Each compute node will run on different machines. There will be at least 4 computes nodes.

For the files (input, intermediate, and output), you can assume that **Client**, **Server** and **Compute Nodes** are sharing the same directory. That is, if a client sends a job with a filename “data.txt”, the server knows where data.txt is located. You may want to make some sub directories e.g., cwd/input_dir, cwd/intermediate_dir, and cwd/output_dir to manage files. Note, CSE VMs for this class use a shared directory via NSF, sharing can be easily done. The input files will contains only number (0 ~ 9999) with a space between number – e.g., `4921 2392 38 3 251 22 1 2354 563 23 … 239 231`

For the chunk data to be sent to compute nodes, the server will send an input filename, offset, and chunk size to let compute nodes read a data chunk from the input file by itself.

Each compute node can expect two kinds of tasks to be executed on it:

1. **Sort**: In this task, the compute node will sort the given chunk data and output the sorted result to an intermediate file (filename will be returned to the server). When the server receives all intermediate filenames for all tasks, i.e., when all sort tasks are done, it will assign **merge** tasks to compute nodes. You will need to make unique names of each intermediate file.
2. **Merge**: In this task, the compute node will take a list of intermediate filenames (a list size is configurable) as an input. The compute nodes will merge those sorted data (from intermediate files) and output a sorted result to an intermediate file (filename will be also returned to the server). The server will keep assigning merge task to compute nodes until it produces the final sorted output. When all merge tasks are done, i.e., when the job is done, the server will return the filename which includes the result (sorted output) to the client.

The server will calculate the number of sort tasks based on the input file size and chunk size for each task when it receives a job from the client. The chunk size will be provided when the server starts as a parameter. If an input file size for a job is 64MB and a chunk size for a task is set to 1MB, there will be 64 sort tasks. You can set the chunk size in terms of number of integers e.g., 1048576, 200000 or any other sizes instead of using MB.

The number of intermediate files for merge tasks is also passed to the server as a parameter. If it is set to 8 and there are 64 sort tasks (intermediated files), there will be 9 merge tasks (8 merge tasks for the first round and 1 merge task for the second round).

Each compute node is responsible for displaying various job statistics for the user such as the number of tasks it received and average time for executed tasks. The server should be able to display statistics like the number of faults which occurred in the system, the number of redundant tasks executed (the server can aggregate this information from all the compute nodes).

2.1. Fault Detection and Recovery:

The system should also be capable of detecting faulty nodes and recovering the task executing on them by re-assigning it to a different compute node. This fault detection can be done either by using a time-out mechanism or by using heartbeat messages. That is, if the server notices that a compute node stops working (is crashed), the server will re-assign tasks previously assigned to that faulty node to other nodes. Any intermediate file generated by the faulty node will be stored without any loss. i.e., only

the tasks which have not been done will be re-assigned. The server will have a job tracker which will be responsible for handling this functionality of the system. The system should be able to guarantee the correct output (all numbers sorted in increasing order) in spite of the failure in one or more compute nodes.

Fault-injection:

The system should be capable of injecting faults, so that it can test the fault detection and recovery mechanism. To do this, each compute node is assigned a 'fail probability' i.e., the probability with which this node will fail while executing the task. It can be assumed that this probability is same for all the nodes and is supplied to the system as a configuration parameter.

2.2. Proactive Fault Tolerance (Extra credit)

In MapReduce framework, node faults are not exception but common which may cause overall performance degradation. To avoid such performance degradation, the server may assign the same tasks to multiple nodes (# configurable) rather than re-assign the task which was executed on failed node to another node. Once framework receives the result from any nodes, it will kill the redundant task(s) executed on other nodes. Thus, even in node failure, the client will see limited performance degradation with redundant task executions. Of course this will bring some cost to run and manage redundant tasks. You will need to show how many redundant tasks have been killed to complete a job.

This is not a mandatory but you can implement it for an extra credit.

2.3 Performance Evaluation

The performance of the system should be evaluated at different 'fail probabilities' e.g., 0%, 10%, 20% and so on, and the effects of the failing nodes on the system performance should be analyzed. For the evaluation, you will vary data size (assuming that max size is 100MB) and chunk size. The results should be plotted for the time taken by the system to complete the sort job under these different 'fail probabilities'. You will also plot of the number tasks launched and how many failed/were killed. If you implement **proactive fault handling**, you are required to measure the performance of the system at different 'fail probabilities'.

3. Implementation Details

Your system may be implemented in C++ or Java using Thrift (or TCP).

The system parameters such as the ‘fail probability’ for compute nodes should be passed using a configuration file or parameter when nodes run.

Some sample input data and expected sorted output will be provided. Thus, you can test your system whether it works well as expected.

Assumptions and Hints:

- The job sent by client can simply be a filename to be sorted.
- You can assume that there will be a single job at a time in order to simplify the system implementation.
- You need to consider how to split input data for the correct output.
- Since each user account on CSE has 1GB of disk space, you will need to remove intermediate files if there is not enough storage space.
- **Please read this document more carefully before you ask.**

4. Project Group

All students should work in groups of size no more than 2 members.

5. Testcases

Basic test cases include testing the system by injecting faults with varying ‘fail probabilities’ (low to high) and ensuring that the output produces is still correct. You must also develop your own test cases, and provide documentation that explains how to run each of your test cases, including the expected results.

6. Deliverables

- Design document describing each component.
- User document explaining how to run each component and how to use the service as a whole, including command line syntax, configuration file particulars, and user input interface.
- Testing description, including a list of cases attempted (which must include negative cases) and the results.
- Source code, Makefiles and/or a script to start the system. (No object (class) file).
- **Only one submission** for each group. (Don’t forget to put all names in a group)

7. Grading

The grade for this assignment will include the following components:

- 40% - The document you submit
    - Description of design and operation of the system - 10%
    - Description of test cases and the performance evaluation results for the system - 30%
- 50% - The functionality and correctness of your program
- 10% - The quality of the source code, in terms of style and in line documentation
- 15% - Extra credit.
    - Implementing proactive fault tolerance - 10%
    - Performance evaluation result with proactive fault tolerance and description of design – 5%

You will lose points if there is any exception, crash or freezing on your programs.

8. References

- Map Reduce: http://research.google.com/archive/mapreduce.html