CSCI 5105: Introduction to Distributed Systems

Spring 2019

Instructor: Abhishek Chandra

Programming Assignment 1: A simple MapReduce-like compute framework

(*Due: Mar/6/2019 – 11:59pm*)

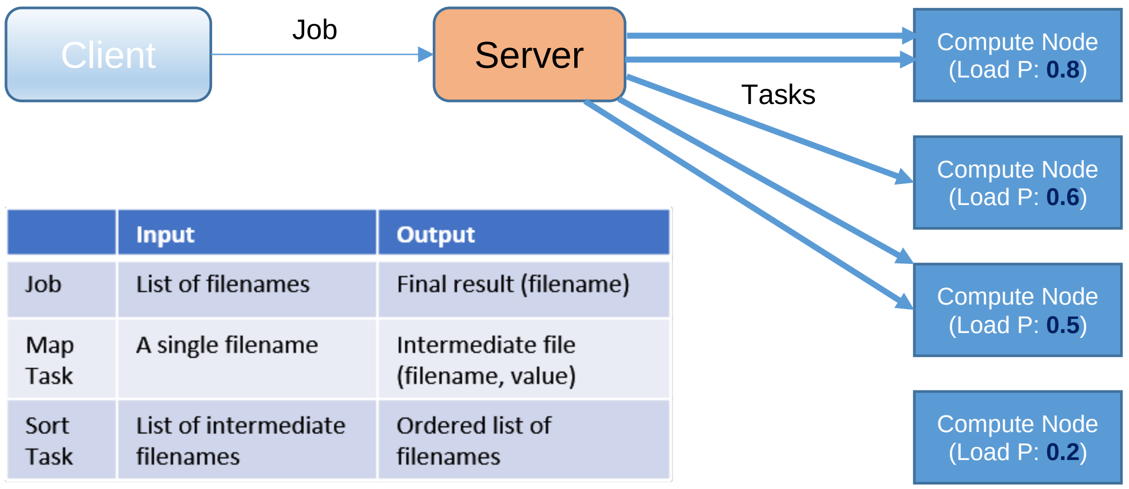
1. Overview

In this programming assignment, you will implement a simple MapReduce-like compute framework for implementing a sentiment analysis program. The framework will receive jobs from a client, split job into multiple tasks, and assign tasks to compute nodes which perform the computations. A user will send a job for sentiment analysis. This framework would use different scheduling policies e.g., Random and Load-balancing, to assign tasks to compute nodes.

2. Project Details

In this project, you will implement 3 components.

1. **Client**: The client sends a job to the server. There will be a single job at a time to simplify the system implementation. The job includes filenames which store data to be analyzed. When the job is done, the client will get a result: an ordered list of filenames based on the sentiment score, and elapsed time for a job.
2. **Server**: The server receives the job submitted by the client. It splits the job into multiple tasks and assigns each task to a compute node. The server needs to log (print) the status of job e.g., the number of running and complete tasks for a job, etc. When the job is completed, the server prints the elapsed time to run it and returns a result to the client.
3. **Compute Nodes**: The compute nodes execute tasks (either map or sort) sent to them by the server. Each compute node will run on a different machine. There must be at least 4 computes nodes in this project.


Client
Job
Server
Tasks
Compute Node (Load P: 0.8)
Compute Node (Load P: 0.6)
Compute Node (Load P: 0.5)
Compute Node (Load P: 0.2)

	Input	Output
Job	List of filenames	Final result (filename)
Map Task	A single filename	Intermediate file (filename, value)
Sort Task	List of intermediate filenames	Ordered list of filenames

In the figure, each compute node is assigned a ‘load-probability’, e.g., 0.8, 0.6, 0.5, and 0.2, that will be explained in next section.

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 list of filenames e.g., 1.txt, 2.txt and so on, the server knows where 1.txt and 2.txt are 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, CSELab machines use a shared directory via NSF, sharing can be easily done. The input files will be text files. The server will assign a task to a compute node by sending a filename for the task.

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

1. **Map**: In this task, the compute node computes a sentiment value of given filename. To get the value, each map task counts positive words and negative words in a given file. Words include only alphabet characters (no punctuation). The list of positive and negative words will be given, i.e., negative.txt and positive.txt. You can ignore words that do not exist in the lists. Each compute node needs to read the list to count positive and negative words. Once the number of positive and negative words are counted, the sentiment value is estimated as below.
   Sentiment = (# positive - # negative) / (# positive + # negative)
   Once a sentiment value is computed, the compute node outputs the result in an intermediate file in key-value format (filename, sentiment value). You will need to make unique names of each intermediate file. The intermediate filename will be returned to the server for a **Sort** task.
2. **Sort**: When the server receives all intermediate filenames for all tasks, i.e., when all map tasks are done, it assigns a sort task to a single designated compute node. In this task, the designated compute node will take a list of intermediate filenames as an input. The compute node will sort the filenames based on sentiment values and output the ordered list in a final output file. Then, compute node returns a result (filename) to the server.

The server calculates the number of tasks based on the number of files when it receives a job from the client. For example, if the number of files for a job is 500, there will be 500 map tasks and 1 sort task.

Each compute node is responsible for displaying various task statistics for the user such as the number of tasks it received, filenames and average time for executed tasks.

Once all tasks are done, the server returns the time taken for a job and final output (filename) to the client which prints these details out on the console for the user.

2.1. Scheduling Policies

In this project, you will need to implement 2 scheduling polices.

1. **Random**: The server will assign tasks to any compute nodes randomly chosen.
2. **Load-Balancing**: The server tries to assign tasks to any compute nodes. However, the compute node may reject accepting a task based on the ‘load-probability’ that will be

explained. If a compute node rejects a task, the server tries to assign the task to other compute nodes until the tasks is accepted.

The scheduling policy will be provided as a parameter when the server starts.

2.2. Load Probability

As mentioned, each compute node is assigned a 'load-probability', e.g., 0.8, 0.6, 0.5, and 0.2. The load-probability is used in two ways.

- Rejecting tasks

In the **Random** scheduling policy, the compute nodes accept and execute tasks as assigned by the server regardless of the load-probability.

In the **Load-balancing** scheduling policy, the compute nodes can reject accepting tasks based on the load-probability when the server tries to assign tasks to it. For example, a compute node has 80% chance to reject tasks if 0.8 load-probability is assigned to the compute node. That is, the compute nodes that have a high load-probability will execute less tasks compared to nodes that has low load-probability.

- Load injecting

The system should be capable of injecting loads, so that it can test the load-balancing scheduling. To this end, each compute node injects a delay (load) based on the load probability before executing tasks regardless of scheduling policies. For example, if 0.8 load-probability is assigned to a compute node, the compute node will have 80% chance to inject delay. You can use *sleep (seconds)* function to simulate the delay before executing tasks. You can use 3 seconds for sleep by default, but it can be changed as you need. Note, each task will be executed by separate threads in compute nodes. So, each thread in each compute node for a task needs to check the load probability individually before running the task and injects delay by calling the sleep function if it is needed.

In short, the load probability is used for 1) rejecting tasks in the Load-balancing scheduling and 2) injecting load regardless of scheduling policies.

2.3. Performance Evaluation

The performance of the system should be evaluated for different task scheduling policies, Random and Load-balancing, and 'load probabilities'.

You can consider scenarios below for load probabilities.

1) All compute nodes have the same (or similar) load probability, e.g., low: (0.2, 0.2, 0.2, 0.2) and high: (0.8, 0.8, 0.8, 0.8).
2) All compute nodes have different load probability, e.g., (0.1, 0.5, 0.2, 0.9).

The effects of load-balancing should be analyzed. The results should be plotted for the time taken by the system to complete the job under these different load probabilities and scheduling policies.

3. Implementation Details

Your system will be implemented in C++ or Java using Thrift.

The server needs to create a tread for each task as tasks are running in parallel. Each compute node must run a multi-threaded Thrift server (**TThreadedServer**) to accept and execute multiple tasks.

The system parameters such as the 'load probability' for each compute node should be passed using a configuration file or parameter when compute nodes run.

Client can send a job using a simple command and pass a directory path that has input files.

```
% java client input_dir
```

Some example input data and expected sentimental scores are included in a given file (data.tar.gz). Thus, you can test your system whether it works well as expected. If you think the expected results are not correct, please let TA know as soon as possible.

Assumptions and Hints:

- There will be a single job at a time to simplify the system implementation.
- The job sent by client can simply be a list of filenames to be analyzed and sorted.
- All communications in this project are synchronous.
- There will be no faulty node during a job execution.
- You can make your own reasonable assumptions if there is something not clear for you. Please specify assumptions you made in the documents.
- **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 load with varying 'load probabilities' (low to high) and ensuring that the output produced 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, including command line syntax, configuration file, or/and user input interface.
- Testing description, including a list of cases attempted (which must include negative cases) and the results.

- Source code, **<u>Makefiles and/or a script</u>** to compile and start the system. (No object file).
- **Only one submission** for each group. (Don't forget to put names & IDs 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

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
- Sentiment Analysis Using MapReduce: https://www.cloudera.com/documentation/other/tutorial/CDH5/topics/ht_example_4_sentiment_analysis.html