Distributed Systems (CSE431) Monsoon 2016
Project Phase-II, MapReduce

Deadline : 8th November, 11:55 PM

Problem Statement:
Implement Distributed grep over a given input file using MapReduce programming paradigm.
MapReduce involves two major components JobTracker(JT) and TaskTracker(TT) and a JobClient which submits a job.

JobClient:
This is invoked from command line with the following args:

<mapName> <reducerName> <inputFile in HDFS> <outputFile in HDFS> <numReducers>

Job:

Define 2 interface classes Mapper (contains map() method) and Reducer (contains reduce() method) , and other helper classes.

You need to implement the two classes <mapName> , <reducerName> which together performs the distributed grep operation. (These class names will be provided as input by the client)

<mapName> will extend the Mapper class and implement the map() method. Similarly <reducerName> will extend the reducer class and implement the reduce() method. This will be packaged as a jar. In C, this will be packaged as a .so.

In both cases, the assumption is that the job jar is available in the TT's classpath (LD_LIBRARY_PATH for C). Before running the job, you will have to manually copy the jar (or .so) to all the nodes running TT.

In Java, use Class.forName() to load the class. In C use dlopen() to load the .so; and dlsym() to invoke the map() and reduce() methods.

The JobClient invokes the following RPCs -

JobSubmitResponse **jobSubmit**(JobSubmitRequest)
JobStatusResponse **getJobStatus**(JobStatusRequest)

JobClient finds the location of JT using a **conf file,** and sends an RPC **jobSubmit** with the protobuf filled accordingly.

On success, the response contains the jobId.
The JobClient then sits in a while loop sending getJobStatus RPCs with the jobId until the JT responds with a completion response.
The client should print progress status to no.of map tasks started, and the no. of reduce tasks started on the console in the following format -

Total Map Tasks : x
Total Map Tasks Started: y
Total Reduce Tasks : <numReducers>
Total Reduce Tasks Started : z

Job Tracker :

The JT assigns an id to the job, and queues the job for processing. It opens the file in HDFS, obtains the block locations to determine the number of map tasks required. You can assume that no.of map tasks is equal to the no.of of blocks in the input hdfs file. The JT then places the tasks in a queue (Map task queue) for the TT to pick up. It uses the locations to decide where to schedule the map tasks.
(Communication overhead is reduced if the task tracker and the datanode that contains the block are running on the same node)

Each TT heartbeats with the JT every second reporting information about how many map/reduce tasks it can run; and also providing status information about the current tasks it is running.

HeartBeatResponse heartBeat(HeartBeatRequest)

When a TT heartbeats, the JT uses the number of map/reduce slots available to decide if it can schedule tasks on the TT. The heartbeat response contains information required to execute the map/reduce tasks.
The heartbeat from TT also contains information about the status of the tasks. JT uses this information to respond to the getJobStatus RPC from the JobClient.

Once all the map tasks for a job are complete, it schedules the reducer(s) on the task tracker nodes based on availability.

Task Tracker :

Each TT heartbeats with the JT every second reporting information about how many map/reduce tasks it can run; and also providing status information about the current tasks it is running.

When the TT gets a map task in heartbeat response, it places the request in an internal queue. The consumers of this queue are a fixed number of threads.

No.of map slots sent by TT in HB = Total no.of map threads - no.of map tasks under execution

(Same goes for reduce tasks)

Each thread picks up a MapTask from the queue. It then instantiates a class of type Mapper by loading the <mapName> Class, and calling its constructor (empty constructor).
It then reads the appropriate HDFS block using readBlock, splits it into lines (\n terminated) and invokes the map() implemented by the class.

The map() will return a string that contains the records ((K,V) pairs) it wants to emit. The helper thread will write the record after adding \n to a file. This file is the output of the map task. Write the output of the map task in a file in HDFS. Give unique names to the map output files like job_<jobid>_map_<taskid>.

The TT then updates the status of this task as completed and sends it in the heartbeat to JT.

Once all the map tasks for a job are complete, the JT will schedule the reducers. To do this, it takes the outputs of the map tasks and divides them among all the reducers (Note that the no.of reducers is provided by job client). The JT then places these tasks in another queue (Reduce task queue) for the TT to pick up.

When the tasktracker gets a reduce task, it queues it up like it does for map tasks. A thread from the reducer thread pool picks up the reducer task and does the following.
- Read each HDFS file. This file contains the records that are emitted by the map task. To simplify, assume that the records as "\n" terminated.
- It invokes the reduce() method with each such line

- The reduce() returns a string. If it is non-null, the string is written as is to the output file.
-Write the output file to HDFS. Give unique names to the reduce output files like <outputfile>_<jobId>_<reducerId>.

Once all the input files are processed for that reduce task, the reducer is complete. The TT then updates the status of this reduce task as completed and sends it in the heartbeat to JT.

JT then checks if all reduce tasks for that job are completed and updates the job status which the job client picks up to mark the job as completed. The final output can be read by reading all the files matching the name <outputfile>_<jobId>_*

Key Points:

- All required proto, java and IDL files have been attached. You can use the same.
- All the parameters and return objects should be marshalled and unmarshalled from byte arrays (Using Google Protobuf)
- Use the same cluster setup you have done for HDFS.