15-440/640 Distributed Systems

Lab 3 Report: Map-Reduce Engine

Team:

Abhishek Bhowmik (abhowmi1),

Neil Rajesh Dhruva (ndhruva)

Date:

11/22/2014

**Contents**

**Problem Definition ………………………………………………………………….. 3**

**Solution Overview …………………………………………………………………… 4**

**Framework ……………………………………………………………………………… 6**

**Code Organization …………………………………………………………………… 11**

**Conclusion ………………………………………………………………………………. 13**

**Problem Definition**

The problem deals with designing a map-reduce framework similar to Hadoop for parallel execution of user tasks. The framework should be able to deploy such tasks in a distributed environment on multiple nodes, with the capacity to start and monitor jobs deployed by a user.

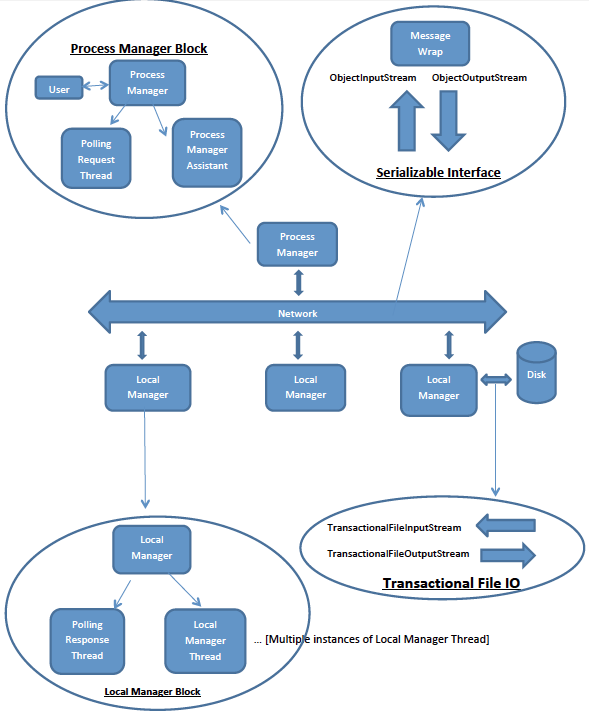
In addition to the parallel execution of tasks, the framework should provide a distributed file system abstraction that keeps track of input and output files corresponding to user jobs. And, while the engine is not required to sort and combine the final output, it should be able to parallelize user jobs efficiently and distribute the work for concurrent execution to improve job latency compared to using a single machine.

The framework should implement an effective scheduler that distributes task load efficiently. Additionally, both the distributed file system, and the map-reduce engine should be capable of recovering from node failure. Additionally, the framework should provide a way for differentiating between records in user files so that users can deal with each record individually through map and reduce functions.

The framework should provide an API for the user to submit tasks, and documentation for the application programmer to effectively use the map-reduce engine. Finally, two examples should be provided in order to demonstrate the functionality of the system.

**Solution Overview**

**Framework**

****

**Distributed File System (with Namenode and Datanodes)**

The distributed file system implemented as part of the map-reduce framework acts as an abstraction for a global file system available across all nodes participating in the framework. It, in many ways, is similar to the way HDFS is implemented in the Hadoop framework.

The key feature of the distributed file system (henceforth, DFS) implemented in this project is how it emulates an actual file system with physical memory. It provides the user with the ability to add files to the DFS, view the structure of the DFS at any given time, delete files present on the DFS and get (download) files from the DFS. The abstraction provided is in the form of representation of the file system, which acts like a regular file system, but distributes the actual data provided by the user across different machines (datanodes) present in the map-reduce engine.

**Working:**

The DFS is implemented on the Namenode, i.e., the master node of the map-reduce framework. The user communicates with it through the Client API. In fact, the client API is the user’s gateway to the map-reduce engine. As will be discussed later, the user adds, downloads and deletes files to and from the DFS using the client API, and also submits map-reduce jobs using the same.

The client API provides the user a clean and simple interface to the DFS by abstracting away all network I/O, and hiding the abstraction provided by the DFS. It also hides the details of how user files are stored across datanodes in the framework. The user deals with files in a manner similar to dealing with local file system files. Hence, the client API, to a great extent, reduces the effort that the user has to put into worrying about processing data in huge files.

The client API itself communicates with the DFS hosted on the namenode using Java’s RMI facility. This makes it very easy for application programmers to add more functionality to the DFS as and when required. All calls to the DFS are through the DFS service hosted on an RMI registry on the namenode. Hence, a major overhead of sockets and network I/O is completely left with Java to deal with. The application programmer can thus focus on the functionality provided by the added features to the DFS, rather than worry about sockets and network messages.

The actual working of the DFS involves converting the user input files to blocks, in order to allow the map-reduce engine to work on smaller blocks of a file, concurrently. Additionally, the DFS maintains a replication factor for each file added by every user. The replication factor ensures that a particular block belonging to a file is available across different datanodes. In short, the number of datanodes that the block is available on is equal to the replication factor.

When the user adds a local file from the user’s file system to DFS, the DFS identifies certain datanodes (according to the replication factor) that can be used to store the files. This is done using a round-robin algorithm to distribute the data across the different nodes. This way, we ensure that the next block goes to a datanode with lesser load (according to space occupied on the datanode) than others. Once the DFS identifies these nodes, it forwards the list to the client API that does the actual work of splitting the user input file into blocks (using an input split criterion for splitting files into blocks of ‘records’), and sending these blocks across the network to the datanodes. The split criterion is provided by the user to be a delimiting string or the number of bytes per record.

Once the input file is split, the blocks are sent as byte streams across the network to the datanodes using RMI. Consequently, each datanode running on a separate machine is provided as a service bound to an RMI registry on that node. The client API communicates with this registry in order to send blocks to the datanodes. Once the file transfer is complete, the user can invoke the map and reduce functionality of the engine.

The additional facilities of file download and deletion are also provided to the user through the client API. When the user requests a file from the DFS, the DFS sends a list of datanodes to the client API which constitute blocks of the file requested by the user. The client API communicates with the datanode RMI services in order to download the corresponding blocks to the local file system of the user. A file deletion command from the user causes the client API to forward the request to the DFS service on the namenode that deletes the file blocks from the datanodes that store these blocks. Finally, the user is also provided the service of printing the current directory structure of the DFS using the appropriate API call on the client API. The DFS does level-order traversal of its structure, described next, and sends it back to the user.

The DFS structure is represented on the namenode using a trie data structure. Each node in the data structure represents a virtual directory that points to zero or more subdirectories, and also points to files stored in that directory. The user always specifies file paths according to the base directory of the DFS. The base directory is the root of the trie. The directory is traversed every time a file is added, requested or deleted, and the corresponding data structures that represent the nodes in the trie, as well as the data structures representing the file itself are accordingly added/altered/deleted. Each data structure that represents a file stores the datanodes on which the blocks of the file are present, which is then used to do operations on the physical blocks stored across the datanodes.

The question that remains, what if a datanode fails? Does the user lose the files uploaded to DFS? The answer is, no. As long as the application programmer maintains a replication factor of greater than 1, there will always be a copy of a file block on more than one node. This is possible because of the round-robin algorithm used for assigning datanodes to store file blocks. The algorithm makes sure that as long as there are more than 1 nodes active, and the replication factor is greater than 1, we have a copy of the block on more than one nodes. However, what if more than 1 node goes down? Well, we have provision for such situations too. Every time a node goes down, it is detected by the job tracker and the namenode/DFS is notified of the same. When this happens, the namenode immediately identifies all the blocks that were present on that node, and all the nodes that have a duplicate copy. These nodes are then asked to replicate the corresponding block(s) to one or more nodes that do not already have the block(s). This way, the replication factor is maintained. And, this is also done in the round robin manner described above. It is important to note that the transfer between datanodes is done entirely by the datanodes, and the namenode is not involved in routing the physical file block form one datanode to another.

However, if the replication factor is more than the number of available datanodes, then only one copy of each block is sent to the various datanodes. This is to avoid unnecessary usage of space on the datanodes. The main reason behind not maintaining multiple copies of the same block on a particular datanode is that most of the time (and the only possibility with our framework as of now) the user only reads from the uploaded file, and does not alter it. Hence, it is not necessary to maintain multiple copies of the same block on a datanode.

Finally, additional features are provided to the DFS like checking the validity of the path on the DFS to add/get/delete a file. Also, we provide the facility of making sure that a block has successfully been received by a datanode, as directed by the namenode/DFS. If the block was supposed to be sent, but was not successfully transferred to a datanode, then the DFS maintains a note of this fact and makes sure that the block is never requested from that datanode.

**Summary**

The DFS is an abstraction for a system-wide file system for the map-reduce framework. It is maintained using different representations for directories and files on the DFS. It is hosted at a central location, the master node (namenode) to be more precise. In order to conduct map-reduce operations on data, the user has to add these files to the DFS. The client API provides various functionalities to the user to add, download and delete files from the DFS.

When the user adds files to the DFS, the DFS distributes the file blocks across various datanodes, and maintains a replication factor for each of the blocks while guaranteeing fault tolerance. Java’s RMI facility is used extensively to communicate between the client, DFS and the datanodes, as well as for transfer of file blocks. And, corresponding API’s are provided, both for the user, and for the application programmer to configure the DFS, and to interact with it in an intuitive manner.

**Conclusion**

The distributed system described above implements a multi-node master/slave framework that caters to a user’s commands of creating and migrating processes between nodes, as well as generating a report of the activity at any given time in the system. While the system works reasonably well according to the test cases that we performed, there are a lot of additions that can be performed (like ‘ack’) to make the system more robust.

However, we believe that this system can deal with requests from a single user efficiently, and also perform efficient and frequent bookkeeping in order to generate useful reports for the user.