Evaluating the Performance of different versions of a Distributed Dictionary Application

Priya Narayana Subramanian
Washington State University
Email: p.narayanasubramani@wsu.edu

***Abstract*—This report discusses the performance analysis of different versions of a distributed dictionary application. The purpose of this study is to explore the possibility of using Hadoop MapReduce framework. MapReduce is a programming model to process large datasets. The programs written in this style are highly scalable, automatically parallelised and can be executed on large clusters of commodity hardware. In this paper I report on the implementation of different versions of a distributed dictionary application using fundamental concepts of distributed systems design concepts like RPC, and MapReduce. I will compare the results obtained using traditional inter process communication protocol (RPC) and the MapReduce algorithm to evaluate the performance benefits of using MapReduce algorithm. The time performance of the three techniques are compared in this study.**

Keywords*—Distributed System, inter-process communication, RPC, Hadoop, MapReduce.

I. Introduction

The Dictionary application written in Java, takes a word from the user and validates the word by checking if the word is available in ”words.txt” file. After the word validation step, the application uses the ”dictionary.txt” file to search for the meaning of the entered word and fetches it’s meaning to the user if it has a meaning in the ”dictionary.txt” file. The application also gives an option for the user to enter a new word and it’s corresponding meaning. The next section discusses about a brief overview of RPC, and MapReduce framework.

A. Remote Procedure Call

In distributed computing a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in another address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.This is a form of clientserver interaction (caller is client, executer is server), typically implemented via a requestresponse message-passing system. The object-oriented programming analog is remote method invocation (RMI). The RPC model implies a level of location transparency, namely that calling procedures is largely the same whether it is local or remote, but usually they are not identical, so local calls can be distinguished from remote calls. Remote calls are usually orders of magnitude slower and less reliable than local calls, so distinguishing them is important.

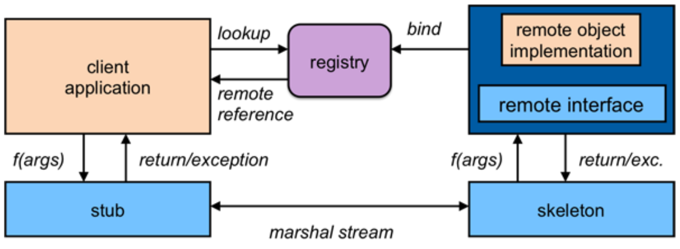


Fig. 1. Java RMI flow

RPCs are a form of inter-process communication (IPC), in that different processes have different address spaces: if on the same host machine, they have distinct virtual address spaces, even though the physical address space is the same; while if they are on different hosts, the physical address space is different. Many different (often incompatible) technologies have been used to implement the concept. In 1995, Sun (the creator of Java) began creating an extension to Java called Java RMI (Remote Method Invocation). Java RMI enables a programmer to create distributed applications where methods of remote objects can be invoked from other Java Virtual Machines (JVMs).

B. MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large datasets. A typical MapReduce computation processes many terabytes of data on thousands of machines. MapReduce usually splits the input dataset into independent chunks. The number of splits depends on the size of the dataset and the number of nodes available. Users specify a map function that processes a (key, value) pair to generate a set of intermediate (key, value) pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

There are separate Map and Reduce steps. Each step done in parallel on sets of (key, value) pairs. Thus, program execution is divided into a Map and a Reduce stage, separated by data transfer between nodes in the cluster. The Map stage takes in a function and a section of data values as input, applies the function to each value in the input set and generates an output set. The Map output is a set of records in the form of (key, value) pairs stored on that node. The records for any given key could be spread across many nodes. The framework, then, sorts the outputs from the Map functions and inputs them into a

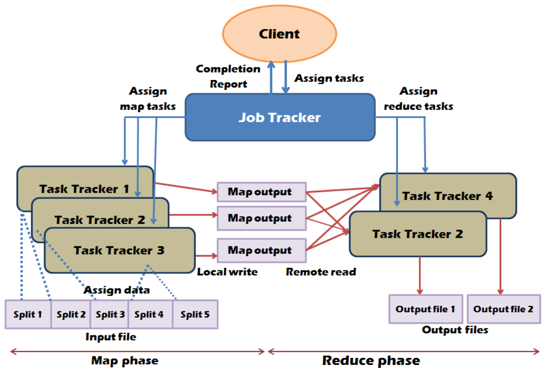


Fig. 2. MapReduce Architecture

Reducer. This involves data transfer between the Mappers and the Reducer. The values are aggregated at the node running the Reducer for that key. The Reduce stage produces another set of (key, value) pairs as final output. The Reduce stage can only start when all the data from the Map stage is transferred to the appropriate machine. MapReduce requires the input as a (key, value)pair that can be serialised and therefore, restricted to tasks and algorithms that use (key, value) pairs.

The MapReduce framework has a single master Job Tracker and multiple Task Trackers. Potentially, each node in the cluster can be a slave Task Tracker. The master manages the partitioning of input data, scheduling of tasks, machine failures, reassignment of failed tasks, inter-machine communications and monitoring the task status. The slaves execute the tasks assigned by the master. Both input and output are stored in the file-system. The single Job Tracker can be a single point failure in this framework. MapReduce is best suited to deal with large datasets and therefore ideal for mining large datasets of petabytes size that do not fit into a physical memory. Most common use of MapReduce is for tasks of additive nature. However, we can tweak it to suit other task.

II. Description of Techniques

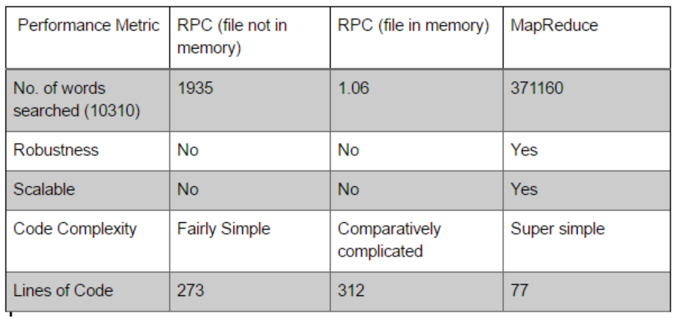
A. In Memory Dictionary

In this section, I will discuss about the different versions of Dictionary implementations. The baseline for comparison is the latest version of dictionary application that has been submitted (RPC). The major drawback of this approach is that it has to read through the file every time it has to serve a request. So, the primary optimization here would be to load the dictionary in memory when the server starts and serve requests from the memory. Also, the previous version did not support writing back to the dictionary. The current implementation supports a dirty write-back where in, it stores the new words and meanings in memory for future requests. when the program exits, it writes these new words and meanings back to the dictionary file so that it can be loaded on the next run. This significantly reduces the number of IO operations thereby providing a huge performance boost.

B. MapReduce

The drawback with these two approaches are that they are not very scalable. For example, if we have a dictionary of size 100 GB or 1 TB, storing it in the memory or reading the files every time is not just impractical but impossible as well. So, I moved to hadoop environment which can handle huge amounts of data. First, the current format of the dictionary file is terrible for map-reduce programs to work on. It's essentially unformatted data. So, I had to do some pre-processing to fix-up the data into a format which can be read by map-reduce programs and pig scripts. Basically, now every word in the dictionary and its meaning will be compacted into a single line, which would help the mapper to split the input along the line boundaries. (For example - IMAGINABILITY : Defn: IMAGINABILITY Im*ag'i*na*bil"i*ty, n. Defn: Capacity for imagination. [R.] Coleridge.). This is a lossy conversion since we lose some newlines but this had to be done for mapreduce to work

I started with one namenode and 4 datanodes all of them being AWS micro instances (1GB RAM and 8GB memory each). This roughly translates into the same hardware capabilities that i had on my single instance. The performance here was not better than storing the dictionary in memory as every run involves a huge setup phase where the data is being allocated, mapped and reduced. But this approach is very scalable as the number of data nodes can be extended to as many as we want and the speedup would be that much. Also, hadoop handles failures and takeovers and hence this is a much robust approach compared to the other two



Performance Metric	RPC (file not in memory)	RPC (file in memory)	MapReduce
No. of words searched (10310)	1935	1.06	371160
Robustness	No	No	Yes
Scalable	No	No	Yes
Code Complexity	Fairly Simple	Comparatively complicated	Super simple
Lines of Code	273	312	77

Fig. 3. The performance evaluation of the 3 versions of distributed dictionary

III. Test Results and Discussion

This section reports the file system settings and results of the measurements. I ran 10310 requests against the RPC server and the total time taken was 1935 seconds. when I ran the same number of requests against the in memory version of the dictionary, it completed in 1.05 seconds. This is understandable since, most of the work is done when the server loads the dictionary onto the hashmap well before the client starts firing the requests (which was not that high either). when i ran a request in hadoop, it took 36 seconds in total to configure the job, collect the results and report it. This is less than ideal, but as we discussed, with high amount of data, this is the only viable and scalable option.

Since, I introduced dirty-write back in the optimized version, there's a observable amount of performance improvement in the in-memory version compared to the vanilla RPC. the speed up was on the lines of read requests.

IV. Conclusion

From my experiments I conclude that, for the given file size, the in-memory dictionary version outperforms the other two versions in terms of execution time. But in terms of Scalability, Robustness and Code complexity the MapReduce version of the dictionary is the winner.