Cluster Based Load Rebalancing in Clouds

S. Aravind Kumar[1], **R. Mangalagowri**[2]

[1]Department of Computer Science, SRM University, Kattankulathur, India
aravind.cse09@gmail.com

[2]Department of Computer Science, Faculty of Engineering and Technology SRM University, Kattankulathur, India
mangalcse@yahoo.com

Abstract: ***Nowadays most of the cloud applications process large amount of data to provide the desired results. Data volumes to be processed by cloud applications are growing much faster than computing power. This growth demands on new strategies for processing and analysing the information. The project explores the use of Hadoop MapReduce framework to execute scientific workflows in the cloud. Cloud computing provides massive clusters for efficient large computation and data analysis. In such file systems, a file is partitioned into a number of file chunks allocated in distinct nodes so that MapReduce tasks can performed in parallel over the nodes to make resource utilization effective and to improve the response time of the job. In large failure prone cloud environments files and nodes are dynamically created, replaced and added in the system due to which some of the nodes are over loaded while some others are under loaded. It leads to load imbalance in distributed file system. To overcome this load imbalance problem, a fully distributed Load rebalancing algorithm has been implemented, which is dynamic in nature does not consider the previous state or behaviour of the system (global knowledge) and it only depends on the present behaviour of the system and estimation of load, comparison of load, stability of different system, performance of system, interaction n between the nodes, nature of load to be transferred, selection of nodes and network traffic. The current Hadoop implementation assumes that computing nodes in a cluster are homogeneous in nature. The performance of Hadoop in heterogeneous clusters where the nodes have different computing capacity is also tested.***

Keywords: Hadoop, Map Reduce, cloud computing, clusters.

1. Introduction

Cloud computing is a relatively new way of referring to the use of shared computing resources, and it is an alternative to having local servers handle applications. Cloud computing groups together large numbers of computer servers and other resources and typically offers their combined capacity on an on-demand, pay-per-cycle basis without sophisticated deployment and management of resources. The end users of a cloud computing network usually have no idea where the servers are physically located, they just spin up their application and start working. This flexibility is the key advantage to cloud computing, and what distinguishes it from other forms of grid or utility computing and software as a service (SaaS). The ability to launch new instances of an application with minimal labor and expense allows application providers to scale up and down rapidly, recover from a failure, bring up development or test instances, and roll out new versions to the customer base.

Distributed file systems are key building blocks for cloud computing applications based on the MapReduce J. Deanet all [1] programming paradigm. MapReduce programs are designed to compute large volumes of data in a parallel fashion. This requires dividing the workload across a large number of machines. Hadoop provides a systematic way to implement this programming paradigm. The computation takes a set of input key/value pairs and produces a set of output key/value pairs. The computation involves two basic operations: Map and Reduce. The Map operation, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate Key #1 and passes them to the Reduce function.

The Reduce function, also written by the user, accepts an intermediate Key #1 and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just an output value of 0 or 1 is produced per Reduce invocation. The intermediate values are supplied to the user's Reduce function via an iterator (an object that allows a programmer to traverse through all the elements of a collection regardless of its specific implementation.

The proposed fully distributed load rebalancing algorithm can be integrated with the Hadoop [3] Single-Node Cluster or Multi-Node Cluster to enhance the performance of the NameNode in balancing the loads of storage nodes present in the cluster. Figure 1 represents a typical Single-Node Hadoop Cluster. The three major categories of machine roles in a Client machines, Masters Nodes, and Slave nodes. The Master nodes oversee the two key functional pieces that make up Hadoop: storing lots of data (HDFS), and running parallel computations on all that data (Map Reduce). The Name Node oversees and coordinates the data storage function (HDFS), while the Job Tracker oversees and coordinates the parallel processing of data using Map Reduce. Slave Nodes make up the vast majority of machines and do all the dirty work of storing the data and running the computations. Each slave runs both a Data Node and Task Tracker daemon that communicate with and receive instructions from their master nodes. The Task Tracker daemon is a slave to the Job Tracker, the Data Node daemon a slave to the Name Node. Client machines have Hadoop installed with all the cluster settings, but are neither a Master nor a Slave. Instead, the role of the Client machine is to load data into the cluster, submit Map Reduce jobs describing how that data should be processed and then retrieve or view the results of the job when it’s finished. In smaller clusters (~40 nodes) you may have a single physical server playing multiple roles, such as both Job Tracker and Name Node.

With medium to large clusters you will often have each role operating on a single server machine.

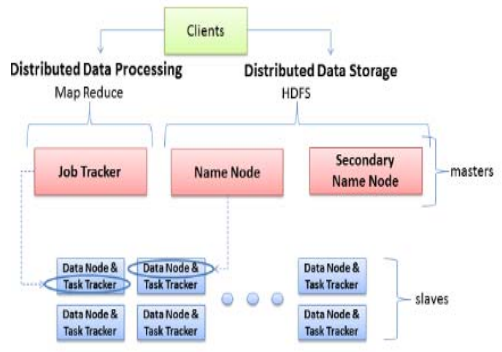

Figure1: Hadoop Server roles in Single Node Cluster

In this paper, we are interested in studying the load rebalancing problem in distributed file systems specialized for large-scale, dynamic and data-intensive clouds. Our objective is to allocate the chunks of files as uniformly as possible among the nodes such that no node manages an excessive number of chunks.

2. Our Proposal

2.1 DHT Architecture

The chunk servers in our proposal are organized as a DHT Network [8]; in such networks file are partitioned into a number of fixed-size chunks, and each chunk is assigned with a unique chunk handle (or chunk identifier) named with a globally known hash function such as SHA1. The hash function returns a unique identifier for a given file's pathname string and a chunk index. The hash function returns a unique identifier for a given file's pathname string and a chunk index. For example, the identifiers of the first and third chunks of file "/user/tom/tmp/a.log" are respectively SHA1 (/user/tom/tmp/a.log, 0) and SHA1 (/user/tom/tmp/a.log, 2).

To discover a file chunk, the DHT lookup operation is performed among the available storage nodes. In most DHTs, the average number of nodes visited for a lookup is O (log n) [1], [2].

In summary, contributions of DHT Architecture are threefold as follows:

- DHTs guarantee that if a node leaves, then its locally hosted chunks are reliably migrated to its successor; if a node joins, then it allocates the chunks whose IDs immediately precede the joining node from its successor to manage, which enables chunk servers to self-configure and self-heal in our proposal,andsimplifying the system provisioning and management.
- In typical DHTs lookups take a modest delay by visiting O (log n) nodes; the lookup latency can be even reduced since discovering the 'l' chunks of a file can also be performed in parallel.
- The DHT network is transparent to the metadata management in our proposal. While the DHT network specifies the locations of chunks precisely and effectively, our proposal can be integrated easily with existing large-scale distributed file systems, e.g., Google GFS (Google File System) [3] and Hadoop HDFS (Hadoop Distributed File System) [5], in which a centralized master node manages the namespace of the entire file system and the mapping of file chunks to storage nodes as metadata information.

2.2. Load Rebalancing Algorithm

a. Overview

A large-scale distributed file system is in a load-balanced state if each chunk server hosts no more than 'A' chunks. In our proposed algorithm, each chunk server node 'I' first estimates whether it is under loaded (light) or overloaded (heavy) without considering the global knowledge. A node is light if the number of chunks it hosts is smaller than the threshold value A. In contrary a Node is heavy if the number of chunks it hosts is more than A. For instance if a node i departs and re-joins as a successor of another node j, then we represent node i as node j + 1, node j's original successor as node j + 2, the successor of node j's original successor as node j + 3, and so on. This process repeats until all the heavy nodes in the system become light nodes. The time complexity of the above algorithm can be reduced if each light node can know which heavy node it needs to request chunks beforehand, and then all light nodes can balance their loads in parallel. Thus, we extend the algorithm by pairing the top-k_1 under loaded nodes with thetop-k_2 overloaded nodes. We use U to denote the set of top-$k_1$underloaded nodes in the sorted list of under loaded nodes, and use O to denote the set of top-k_2 overloaded nodes in the sorted list of overloaded nodes.

Based on the above-introduced load balancing algorithm, the light node that should request chunks from the k2-th most loaded node in O is the k1thleast loaded node in U is identified easily and load between them is shared accordingly.

b. Basic Algorithms

In the basic algorithm, each node implements the gossip-based aggregation protocol in [26], [27] to collect the load statuses of a sample of randomly selected nodes. Specifically, each node contacts a number of randomly selected nodes in the system and builds a vector denoted by V.

A vector consists of entries, and each entry contains the ID, network address and load status of a randomly selected node. Using the gossip-based protocol, each node i exchanges its locally maintained vector with its neighbours until its vector has s entries. It then calculates the average load of the s nodes denoted by A.

If node i finds itself is a light node, it seeks a heavy node to request chunks. Node i sorts the nodes in its vector including itself based on the load status and finds its position k1 in the sorted list, i.e., it is the top-k1 under loaded node in the list. Algorithm 1 specifies the operation that a light node i seeks an overloaded node j, and Algorithm 2 shows that i requests some file chunks from j.

2.2 Chunk Creation

A file is partitioned into a number of chunks allocated in distinct nodes so that Map Reduce Tasks can be performed in parallel over the nodes. The load of a node is typically proportional to the number of file chunks the node possesses.

Files and nodes in a cloud environment can be arbitrarily created, deleted, and appended, and nodes can be upgraded, replaced and added in the file system, the file chunks are not distributed as uniformly as possible among the nodes. The main objective is to allocate the chunks of files as uniformly as possible among the nodes such that no node manages an excessive number of chunks.

2.2 DHT Formulation

The storage nodes are structured as a network based on distributed hash tables(*DHTs*), e.g., discovering a file chunk can simply refer to rapid key lookup in DHTs, given that a unique handle (or *identifier*) is assigned to each file chunk.

DHTs enable nodes to self-organize and repair while constantly offering lookup functionality in node dynamism, simplifying the system provision and management. The chunk servers in are organized as a DHT network.

Typical DHTs guarantee that if a node leaves, then its locally hosted chunks are reliably migrated to its successor; if a node joins, then it allocates the chunks whose IDs immediately precede the joining node from its successor to manage.

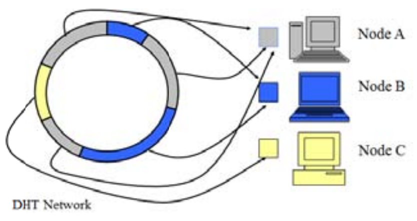

Figure 2. An example depicts the distribution of storage nodes in a DHT Network along the load statuses of each storage nodes.

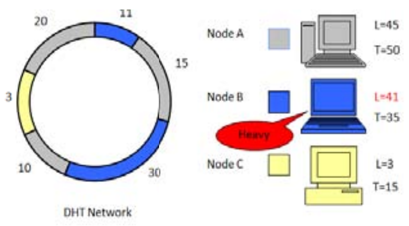

Figure 3. An DHT Network in which the Load of Node B is Heavy, since the more Load is assigned to it ,where 'T' represent the Target load(maximum Load assigned to Nodes) of each storage Node and 'L' represent the Load of each Node.

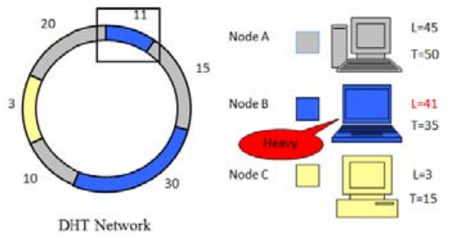

Figure 4. Load status of Node B is assumed to be Heavy, which makes the system to move into a Load imbalanced state leading to system failure.

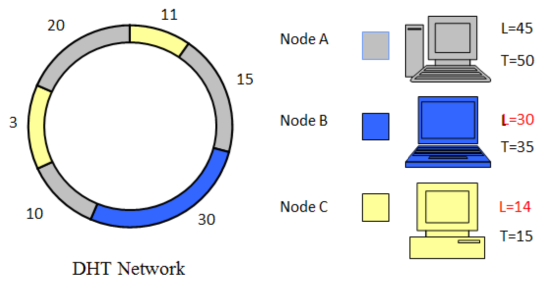

Figure 5. Heavy Load assigned to Node B is dynamically rebalanced (Node C) using the proposed Load Rebalancing Algorithm to the make the system Fully Load Balanced which enhances the overall system performance which is highly desired in a large-scale, data-intensive clouds.

2.3 Load Rebalancing Algorithm

In the Load rebalancing algorithm, each chunk server node (Fig 3.1) first estimate whether it is under loaded (light) or overloaded (heavy) without global knowledge. A node is *light* if the number of chunks it hosts is smaller than the threshold.

Load statuses of a sample of randomly selected nodes. Specifically, each node contacts a number of randomly selected nodes in the system and builds a vector denoted by V. A vector consists of entries, and each entry contains the ID, network address and load status of a randomly selected node.Algorithms1 and 2 are executed simultaneously without synchronization.

2.4 Replica Management

In distributed file systems (e.g., Google GFS and Hadoop HDFS), a constant number of replicas for each file chunk are maintained in distinct nodes to improve file availability with respect to node failures and departures.

Load rebalancing algorithm does not treat replicas distinctly. It is unlikely that two or more replicas are placed in an identical node because of the random nature of our load rebalancing algorithm. More specifically, each under loaded node samples a number of nodes, each selected with a probability of 1/n, to share their loads (where n is the total number of storage nodes).

3. Conclusion

A novel load balancing algorithm to deal with the load rebalancing problem in large-scale, dynamic, and distributed file systems in clouds has been presented in this paper. Our proposal strives to balance the loads of nodes and reduce the demanded movement cost as much as possible, while taking advantage of physical network locality and node heterogeneity. In the absence of representative real workloads in the public domain, we have investigated the performance of our proposal and compared it against competing algorithms through synthesized probabilistic distributions of file chunks.

The synthesis workloads stress test the load balancing algorithms by creating a few storage nodes that are heavily loaded. Our proposal is comparable to the centralized algorithm in HDFS(Hadoop distributer file system and can be incorporated in a Single-Node or Multi-Node Hadoop HDFS cluster environment thorough which the clustering of the Storage Nodes can be done easily and also helps in cluster Node's provisioning and management in Clouds. Hence maintaining the cluster environment in a Load balanced state even if load of system is increased linearly, since the nature of our algorithm is fully distributed and dynamic .The proposed load balancing algorithm dramatically outperforms the competing distributed algorithm in terms of load imbalance factor, movement cost, and algorithmic overhead and also in a proximity-aware manner among the storage Nodes.

4. Future Work

In complex and large systems, there is a tremendous need for load balancing. For simplifying load balancing globally (e.g. in a cloud), one thing which can be done is, employing techniques would act at the components of the clouds in such a way that the load of the whole cloud is balanced. Cloud Computing is a vast concept and load balancing plays a very important role in case of Clouds. There is a huge scope of improvement in this area. The performance of the given algorithms can also be increased by varying different parameters.

5. Acknowledgment

I express my sincere thanks to Mrs. R. MangalaGowri, Assistant Professor and Dr. D. Malathi Head of the Department for providing guidance during the course of this work.

References

[1] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," in Proc. ACM SIGCOMM'04, Aug. 2004, pp. 353–366.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proc. 6th Symp. Operating System Design and Implementation (OSDI'04), Dec. 2004, pp. 137–150.
[3] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," in Proc. 19th ACM Symp. Operating Systems Principles (SOSP'03), Oct.2003, pp. 29–43.
[4] Apache Hadoop, http://hadoop.apache.org/.
[5] Hadoop Distributed File System, http://hadoop.apache.org/hdfs/.
[6] Hadoop Distributed File System, "Rebalancing Blocks,"
[7] http://developer.yahoo.com/hadoop/tutorial/module2.html#rebalancing.
[8] Y. Zhu and Y. Hu, "Efficient, Proximity-Aware Load Balancing for DHT based P2P Systems," IEEE Trans. Parallel Distrib. Syst., vol. 16, no. 4, pp. 349–361, Apr. 2005.
[9] VMware, http://www.vmware.com/Xen, http://www.xen.org/