Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 2, Issue. 12, December 2013, pg.45 – 49

SURVEY ARTICLE

Survey on Novel Load Rebalancing for Distributed File Systems

Suriya Mary.M[1], Guru Rani.G[2]
PG Student[1], Assistant Professor[2], Department of CSE
NPR college of Engineering and Technology, TamilNadu, India
Email: suriyamary.mc@gmail.com; ranitcguru@gmail.com

Abstract- Distributed file system is simply a classical model of a file system used as the key building blocks for cloud computing applications. In such file systems a file is partitioned into a number of chunks allocated in distinct nodes. Files can be dynamically created, deleted, and appended. This results in load imbalance in a distributed file system; that is, the file chunks are not distributed as uniformly as possible among the nodes. Emerging distributed file systems in production systems strongly depend on a central node for chunk reallocation. In this survey paper, a fully distributed load rebalancing algorithm is presented to overcome the above load imbalance problem. To eliminate the dependence on storage nodes each node performs the load rebalancing algorithm independently without acquiring global knowledge.

Keywords- Load balance; distributed file systems; clouds

I. INTRODUCTION

Cloud computing refers to applications and services offered over the Internet. These services are offered from data centers all over the world, which collectively are referred to as the "cloud." This metaphor represents the intangible, yet universal nature of the Internet. The idea of the "cloud" simplifies the many network connections and computer systems involved in online services. In fact, many network diagrams use the image of a cloud to represent the Internet. This symbolizes the Internet's broad reach, while simplifying its complexity. Any user with an Internet connection can access the cloud and the services it provides. Since these services are often connected, users can share information between multiple systems and with other users. Cloud Computing is a compelling technology. In clouds, clients can dynamically allocate their resources on-demand without sophisticated deployment and management of resources. Key enabling technologies for clouds include the MapReduce programming paradigm, distributed file systems, virtualization and so forth. These techniques emphasize scalability, so clouds can be large in scale, and comprising entities can arbitrarily fail and join while maintaining system reliability.

Distributed file systems are key building blocks for cloud computing applications based on the MapReduce programming paradigm. In such file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed in parallel over the nodes. For example, consider a word count application that counts the number of distinct words and the frequency of each unique word in a large file. In such an application, a cloud partitions the file into a large number of disjointed and fixed-size pieces (or file chunks) and assigns them to

different cloud storage nodes (i.e., chunkservers). Each storage node (or node for short) then calculates the frequency of each unique word by scanning and parsing its local file chunks. In distributed file system, the load of a node is typically proportional to the number of file chunks the node possesses. Load balance among storage nodes is a critical function in clouds. In a load-balanced cloud, the resources can be well utilized and provisioned, maximizing the performance of MapReduce-based applications. When the number of storage nodes, the number of files and the number of accesses to files increase linearly, the central nodes (e.g., the master in Google GFS) become a performance bottleneck, as they are unable to accommodate a large number of file accesses due to clients and MapReduce applications. Thus depending on the central nodes to tackle the load imbalance problem exacerbates their heavy loads. Even with the latest development in distributed file systems, the central nodes may still be overloaded. For example, HDFS federation suggests architecture with multiple namenodes (i.e., the nodes managing the metadata information). Its file system namespace is statically and manually partitioned to a number of namenodes. However, as the workload experienced by the namenodes may change over time and no adaptive workload consolidation and/or migration scheme is offered to balance the loads among the namenodes, any of the namenodes may become the performance bottleneck. In this survey paper, the load rebalancing problem in distributed file systems specialized for large-scale, dynamic and data-intensive clouds. Such a large-scale cloud has hundreds or thousands of nodes (and may reach tens of thousands in the future).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.

To eliminate the dependence on central nodes by offloading the load rebalancing task to storage nodes by having the storage nodes balance their loads spontaneously. The storage nodes are structured as a network based on distributed hash tables (DHTs), 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. Additionally, we aim to reduce network traffic (or *movement cost*) caused by rebalancing the loads of nodes as much as possible to maximize the network bandwidth available to normal applications. Each chunk server node first estimates whether it is underloaded (light) or overloaded (heavy) without global knowledge. The numbers of file chunks are migrated to balance the loads of nodes. The same process is executed to release the extra load on the next heaviest node in the system. This process repeats until all the heavy nodes in the system become light nodes. To overcome the above load balancing problem each node performs the load rebalancing algorithm independently without acquiring global knowledge.

II. SOME OF THE FRAMEWORKS

The Apache Foundation's Hadoop Distributed File System (HDFS) and MapReduce engine comprise a distributed computing infrastructure inspired by Google MapReduce and the Google File System (GFS). The Hadoop framework allows processing of massive data sets with distributed computing techniques by leveraging large numbers of physical hosts. Hadoop's use is spreading far beyond its open source search engine roots. The Hadoop frame-work is also being offered by "Platform as a Service" cloud computing providers. Hadoop is made up of two primary components. These components are the Hadoop Distributed File System (HDFS) and the MapReduce engine. HDFS is made up of geographically distributed Data Nodes. Access to these Data Nodes is coordinated by a service called the Name Node. Data Nodes communicate over the network in order to rebalance data and en-sure data is replicated throughout the cluster. The MapReduce engine is made up of two main components. Users submit jobs to a Job Tracker which then distributes the task to Task Trackers as physically close to the required data as possible. While these are the primary components of a Hadoop cluster there are often other services running in a Hadoop cluster such as a workflow manager.

Amazon Elastic MapReduce

Amazon Elastic MapReduce (Amazon EMR) is a web service that makes it easy to quickly and cost-effectively process vast amounts of data. Amazon EMR uses Hadoop, an open source framework, to distribute your data and processing across a resizable cluster of Amazon EC2 instances. Amazon EMR is used in a variety of applications, including log analysis, web indexing, data warehousing, machine learning, financial analysis, scientific simulation, and bioinformatics. Customers launch millions of Amazon EMR clusters every year.

III. LITERATURE SURVEY

3.1MapReduce: Simplified Data Processing on Large Clusters

MapReduce is a programming model and an associated implementation for processing and generating large data sets. 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. Many real world tasks are expressible in this model.The map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately. The functional model with user specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance. Programs written in this functional style are automatically parallelized and

executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Distributed file systems are key building blocks for cloud computing applications based on the MapReduce programming paradigm. In such file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed in parallel over the nodes. The implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

The MapReduce programming model has been successfully used at Google for many different purposes. The model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google. So MapReduce processes many terabytes of data on thousands of machines. Easy to use scalable programming model for large-scale data processing on clusters. It achieves efficiency through disk-locality and also achieves fault-tolerance through replication.

3.2The Google File System

A scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.. A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunklease management, garbage collection of orphaned chunks, and chunkmigration between chunkservers. The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state.

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware .The system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

3.3 Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems

DHT based P2P systems offer a distributed hash table (DHT) abstraction for object storage and retrieval. Many solutions have been proposed to tackle the load balancing issue in DHT-based P2P systems. However, all these solutions either ignore the heterogeneity nature of the system, or reassign loads among nodes without considering proximity relationships, or both.

To tackle this issue an efficient, proximity-aware load balancing scheme by using the concept of virtual servers. The goal is to ensure fair load distribution over nodes proportional to their capacities, but also to minimize the load-balancing cost (e.g., bandwidth consumption due to load movement) by transferring virtual servers between heavily loaded nodes and lightly loaded nodes in a proximity-aware fashion.

To achieve the latter goal by using proximity information to guide virtual server reassignments. There are two main advantages of a proximity-aware load balancing scheme. First, from the system perspective, a load balancing scheme bearing network proximity in mind can reduce the bandwidth consumption (e.g., bisection backbone bandwidth) dedicated to load movement. Second, it can avoid transferring loads across high-latency wide area links, thereby enabling fast convergence on load balance and quick response to load imbalance.

To use proximity information in load balancing the main contributions are: 1) Relying on a self-organized, fully distributed k-ary tree structure constructed on top of a DHT, load balance is achieved by aligning those two skews in load distribution and node capacity inherent in P2P systems. 2) Proximity information is used to guide virtual server reassignments such that virtual servers are reassigned and transferred between physically close heavily loaded nodes and lightly loaded nodes, thereby minimizing the load movement cost and allowing load balancing to perform efficiently.

3.4Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

A distributed peer-to-peer applications need to determine the node that stores a data item. The Chord protocol solves this challenging problem in decentralized manner. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps. Chord simplifies the design of peer-to-peer systems and applications based on it by addressing these difficult problems:

Load balance: Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.

Decentralization: Chord is fully distributed: no node is more important than any other. This improves robustness and makes Chord appropriate for loosely-organized peerto- peer applications.

Scalability: The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible. No parameter tuning is required to achieve this scaling.

Availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, barring major failures in the underlying network, the node responsible for a key can always be found. This is true even if the system is in a continuous state of change.

Flexible naming: Chord places no constraints on the structure of the keys it looks up: the Chord key-space is flat. This gives applications a large amount of flexibility in how they map their own names to Chord keys.

Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them. Chord assigns keys to nodes with *consistent hashing* which has several desirable properties. With high probability the hash function balances load (all nodes receive roughly the same number of keys). Also with high probability, when an Nth node joins (or leaves) the network, only a O(1=N) fraction of the keys are moved to a different location—this is clearly the minimum
necessary to maintain a balanced load. Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node. A Chord node needs only a small amount of "routing" information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with other nodes. In an N-node network, each node maintains information about only O(logN) other nodes, and a lookup requires O(logN) messages.

The consistent hash function assigns each node and key an m bit *identifier* using SHA-1 [10] as a base hash function. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. We will use the term "key" to refer to both the original key and its image under the hash function, as the meaning will be clear from context. Similarly, the term "node" will refer to both the node and its identifier under the hash function. The identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. Consistent hashing assigns keys to nodes as follows. Identifiers are ordered on an *identifier circle* modulo 2m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor*(k). If identifiers are represented as a circle of numbers from 0 to 2m- 1, then *successor*(k) is the first node clockwise from k.

In this paper the Chord uses consistent hashing to assign keys to Chord nodes. Consistent hashing tends to balance load, since each node receives roughly the same number of keys, and requires relatively little movement of keys when nodes join and leave the system. Chord will be a valuable component for peer-to-peer, large-scale distributed applications and also adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Chord acts as a distributed hash function, spreading keys evenly over the nodes; this provides a degree of natural load balance.

3.5Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems

A new framework, called Histogram-based Global Load Balancing (HiGLOB) to facilitate global load balancing in structured P2P systems. Each node P in HiGLOB has two key components. The first component is a histogram manager that maintains a histogram that reflects a global view of the distribution of the load in the system. The histogram stores statistical information that characterizes the average load of no overlapping groups of nodes in the P2P network. It is used to determine if a node is normally loaded, overloaded, or under loaded. The second component of the system is a load balancing manager that takes actions to redistribute the load whenever a node becomes overloaded or under loaded. The load-balancing manager may redistribute the load both statically when a new node joins the system and dynamically when an existing node in the system becomes overloaded or under loaded.

While histograms are useful, the cost of constructing and maintaining them may be expensive especially in dynamic systems. As a result, we introduce two techniques that reduce the maintenance cost. . Reduce the cost of constructing histogram. Constructing a histogram for a new node may be expensive since it requires histogram information from all neighbor nodes. Additionally, the histograms of the new node's neighbors also need to be updated since adding a new node to a group of nodes changes the average load of that group. Constructing and maintaining histograms may therefore be expensive if nodes join and leave the system frequently. In light of the fact that every new node in the P2P system must find and notify its neighbor nodes about its existence while these neighbor nodes need to send their information to the new node to setup connections after that, we suggest that histogram information can be piggybacked with messages used in this process. In this way, we can avoid sending separate histogram messages and totally eliminate the effect of node join on the histogram construction of the new node and histogram update of its neighbor nodes. The overhead cost of using histograms is now solely based on histogram update messages caused by changing of load at nodes in the system.

Maintaining histograms can be expensive since any load change at a node causes an update to be propagated to all other nodes in the system. To avoid this propagation, we suggest that we do not need to keep exact histogram values. We only need to keep approximate values in the histogram. A node only needs to send load information to other nodes when there is a significant change in either its load or the average load of a non-overlapping group in its histogram.

The proposed a framework, HiGLOB, to enable global load balance for structured P2P systems. Each node in HiGLOB maintains the load information of nodes in the systems using histograms. This enables the system to have a global view of the load distribution and hence facilitates global load balancing. To partition the system into non-overlapping groups of nodes and maintain the average load of them in the histogram at a node. The proposed techniques are used to reduce the overhead of maintaining and constructing histograms.

IV. CONCLUSION

In this paper we have done literature survey for analyzing the load imbalance problem for distributed file system. This is our first paper in which only the overview of load rebalancing algorithm have been done and we will provide a load balanced cloud, then only the resources can be well utilized and provisioned, maximizing the performance of MapReduce-based applications. The 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. The proposal strives to balance the loads of data nodes and task nodes efficiently. Then only can able to distribute the file chunks as uniformly as possible. The proposed algorithm operates in a distributed manner in which nodes perform their load-balancing tasks independently without synchronization or global knowledge regarding the system. In a load-balanced cloud, the resources can be well utilized and provisioned, maximizing the performance of MapReduce-based applications. The algorithm also outperforms the competing distributed algorithm in terms of load imbalance factor, movement cost, and algorithmic overhead.

REFERENCES

[1] Hung-Chang Hsiao, Member, IEEE Computer Society, Hsueh-Yi Chung,Haiying Shen, Member, IEEE, and Yu-Chang Chao, proposed a "Load Rebalancing for Distributed File Systems in Clouds" IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 24, NO. 5, MAY 2013

[2]J.Dean and S. Ghemawat,proposed a "MapReduce: Simplified Data Processingon 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," Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03), pp. 29-43, Oct. 2003.

[4] Y. Zhu and Y. Hu, proposed a "Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems," IEEE Trans. Parallel and Distributed Systems, vol. 16, no. 4, pp. 349-361, Apr. 2005.

[5] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan,proposed a "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," IEEE/ACM Trans. Networking, vol. 11, no. 1, pp. 17-21, Feb. 2003.

[6] Q.H. Vu, B.C. Ooi, M. Rinard, and K.-L. Tan, proposed a "Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems," IEEE Trans. Knowledge Data Eng., vol. 21, no. 4, pp. 595-608, Apr. 2009.

[7]A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS '02), pp. 68-79, Feb. 2003.

