IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727 Volume 13, Issue 2 (Jul. - Aug. 2013), PP 60-65
www.iosrjournals.org

Simple Load Rebalancing For Distributed Hash Tables In Cloud

Ch. Mounika[1], L. RamaDevi[2], P.Nikhila[3]
[1]*M.Tech (S.E), VCE, Hyderabad, India,*
[2]*M.Tech (S.E), VCE, Hyderabad, India,*
[3]*M.Tech (S.E), VCE, Hyderabad, India,*

Abstract: *Distributed file systems are key building blocks for cloud computing applications based on the Map Reduce 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 Map Reduce tasks can be performed in parallel over the nodes. However, in a cloud computing environment, failure is the norm, and nodes may be upgraded, replaced, and added in the system. This dependence is clearly inadequate in a large-scale, failure-prone environment because the central load balancer is put under considerable workload that is linearly scaled with the system size, and may thus become the performance bottleneck and the single point of failure. In this paper, a fully distributed load rebalancing algorithm is presented to cope with the load imbalance problem.*

Our algorithm is compared against a centralized approach in a production system and a competing distributed solution presented in the literature. The simulation results indicate that our proposal is comparable with the existing centralized approach and considerably outperforms the prior distributed algorithm in terms of load imbalance factor, movement cost, and algorithmic overhead.

Keywords: *DHT, CentraliseSystem, LoadImBalancing, Distributed System*

I. Introduction

The Distributed file systems an important issue in DHTs is load-balance the even distribution of items to nodes in the DHT. All DHTs make some effort to load balance; generally by randomizing the DHT address associated with each item with a "good enough" hash function and making each DHT node responsible for a balanced portion of the DHT address space. Chord is a prototypical example of this approach: its "random" hashing of nodes to a ring means that each node is responsible for only a small interval of the ring address space, while the random mapping of items means that only a limited number of items land in the (small) ring interval owned by any node. The cloud computing Current distributed hash tables do not evenly partition the address space into.

Which keys get mapped; some machines get a larger portion of it. Thus, even if keys are numerous and random, some machines receive more than their fair share, by as much as a factor of n times the average. To cope with this problem, many DHTs use virtual nodes each real machine pretends to be several distinct machines, each participating independently in the DHT protocol. The machine's load is thus determined by summing over several virtual nodes', creating a tight con cent ration of load near the average. As an example, the Chord DHT is based upon consistent hashing which requires virtual copies to be operated for every node.

The node will occasionally check its inactive virtual nodes, and may migrate to one of them if the distribution of load in the system has changed. Since only one virtual node is active, the real node need not pay the original Chord protocol's multiplicative increase in space and band width costs. Our solution to this problem therefore allows nodes to move to arbitrary addresses; with this freedom we show that we can load balance an arbitrary distribution of items, without expending much cost in maintaining the load balance. Our scheme works through a kind of "work stealing" in which under loaded nodes migrate to portions of the address space occupied by too many items. The protocol is simple and practical, with all the complexity in its performance analysis. 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. Lastly, permitting nodes to choose arbitrary addresses in our item balancing protocol makes it easier for malicious nodes to disrupt the operation of the P2P network. It would be interesting to find counter-measures for this problem.

The paper is organized as follows. Section II Related work, Section III. System Model Section IV Load balancing algorithm. Section V Distributed files system. Section VI. . Performance Evaluation VII concludes.

II. Related Work

This attempt to load-balance can fail in two ways. First, the typical "random" partition of the address space among nodes is not completely balanced. Some nodes end up with a larger portion of the addresses and thus receive a larger portion of the randomly distributed items. Second, some applications may preclude the

randomization of data items' addresses. For example, to support range searching in a database application the items may need to be placed in a specific order, or even at specific addresses, on the ring. In such cases, we may find the items unevenly distributed in address space, meaning that balancing the address space among nodes is not adequate to balance the distribution of items among nodes. We give protocols to solve both of the load balancing challenges just described.

Performance in a P2P System:

Our online load balancing algorithms are motivated by a new application domain for range partitioning peer-to-peer systems. P2P systems store a relation over a large and dynamic set of nodes, and support queries over this relation. Many current systems, known as Distributed Hash Tables (DHTs) use hash partitioning to ensure storage balance, and support point queries over the relation. There has been considerable recent interest in developing P2P systems that can support efficient range queries. For example, a P2P multi-player game might query for all objects located in an area in a virtual 2-D space. In a P2P web cache, a node may request (pre-fetch) all pages with a specific URL prefix. It is well-known that hash partitioning is inefficient for answering such ad hoc range queries, motivating a search for new networks that allow range partitioning while still maintaining the storage balance offered by normal DHTs.

Handling Dynamism in the Network:

The network is a splits the range of Nh to take over half the load of Nh, using the NBRADJUST operation. After this split, there may be NBRBALANCE violations between two pairs of neighbors and In response, ADJUSTLOAD is executed, first at node Nh and then at node N. It is easy to show (as in Lemma 3) that the resulting sequence of NBRADJUST operations repair all NBRBALANCE violations.

Node Departure:

While in the network, each node manages data for a particular range. When the node departs, the data is stored becomes unavailable to the rest of the peers. P2P networks reconcile this data loss in two ways: (a) Do nothing and let the "owners" of the data deal with its availability. The owners will frequently poll the data to detect its loss and re-insert the data into the network.

Maintain replicas of each range across multiple nodes. A Skip Net DHT organizes peers and data objects according to their lexicographic addresses in the form of a variant of a probabilistic skip list. It supports logarithmic time range-based lookups and guarantees path locality. Mercury is more general than Skip Net since it supports range-based lookups on multiple-attributes. Our use of random sampling to estimate query selectivity constitutes a novel contribution towards implementing scalable multi-dimensional range queries. Load balancing is another important way in which Mercury from Skip Net. While Skip Net incorporates a constrained load-balancing mechanism, it is only useful when part of a data name is hashed, in which case the part is inaccessible for performing a range query. This implies that Skip Net supports load-balancing or range queries not both.

III. System Model

3.1 Data Popularity:

Unfortunately, in many applications, a particular range of values may exhibit a much greater popularity in terms of database insertions or queries than other ranges. This would cause the node responsible for the popular range to become overloaded. One obvious solution is to determine some way to partition the ranges in proportion to their popularity. As load patterns change, the system should also move nodes around as needed.

We leverage our approximate histograms to help implement load-balancing in Mercury. First, each node can use histograms to determine the average load existing in the system, and, hence, can determine if it is relatively heavily or lightly loaded. Second, the histograms contain information.

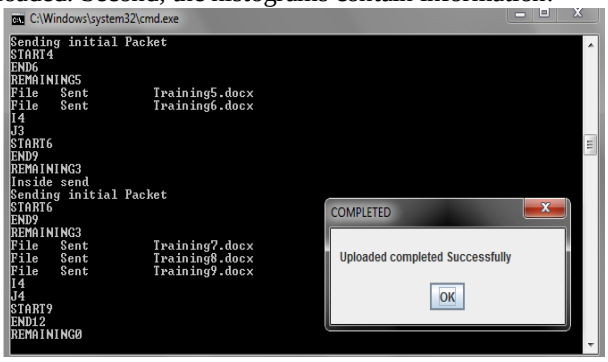

Fig.1 About which parts of the overlay are lightly loaded.

3.2 Load Balancing:

We have shown how to balance the address space, but sometimes this is not enough. Some applications, such as those aiming to support range-searching operations, need to specify a particular, non-random mapping of items into the address space. In this section, we consider a dynamic protocol that aims to balance load for arbitrary item distributions. To do so, we must sacrifice the previous protocol's restriction of each node to a small number of virtual node locations instead, each node is free to migrate anywhere. Our protocol is randomized, and relies on the underlying P2P routing framework only insofar as it has to be able to contact "random" nodes in the system (in the full paper we show that this can be done even when the node distribution is skewed by the load balancing protocol). The protocol is the following, to state the performance of the protocol, we need the concept of a half-life [LNBK02], which is the time it takes for half the nodes or half the items in the system to arrive or depart.

3.3 DHT Implementation

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 our proposal 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. Now we describe the application of this idea to DHTs. Let h0 be a universally agreed hash function that maps peers onto the ring. Similarly, let h1; h2; : : :hd be a series of universally agreed hash functions mapping items onto the ring. To insert an item x using d hash functions, a peer calculates h1(x); h2(x); : : : ; hd(x). Then, d lookups are executed in parallel to and the peers p1; p2; : : : ; pd responsible for these hash values, according to the mapping given by h0,

3.4 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. Because the files in a cloud 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. 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.

Fig.2 Chunk creation

3.5 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. Our current load balancing 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).

IV. Load Balancing Algorithm

In our proposed algorithm, each chunk server node I 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.

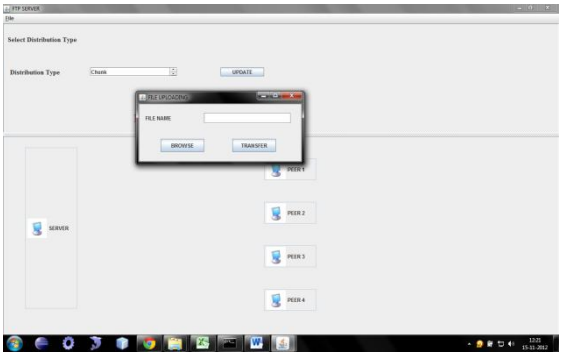

Fig.3 Load Balancing

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. Fig. 3 shows the total number of messages generated by a load rebalancing algorithm.

Load-balanced state:
If each chunk server hosts no more than Am chunks. In our proposed algorithm, each chunk server node I first estimates whether it is under loaded (light) or overloaded (heavy) without global knowledge. Lj A from j to relieve j's load. Node j may still remain as the heaviest node in the system after it has migrated its load to node i. In this case, the current least-loaded node, say node i departs and then rejoins the system as j's successor. That is, I become node j+1, and j's original successor i thus becomes node j + 2. Such a process repeats iteratively until j is no longer the heaviest. Then, 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.

Others: We will offer a rigorous performance analysis for the effect of varying nV in Appendix E. Specifically, we discuss the tradeoff between the value of nV and the movement cost. A larger nV introduces more overhead for message exchanges, but results in a smaller movement cost.

Procedure 1 ADJUSTLOAD (Node Ni)

On Tuple Insertg
1 : *Let* $L(Ni) = x$ 2 (*Tm*; $Tm+1$)
2 : *Let Nj be the lighter loaded of Ni* ?1 *and Ni* +1.
3 : **if***L* (Nj) _ *Tm* ?1**then***fDo NBRADJUSTg*
4: Move tuples from Ni to Nj to equalize load.
5: ADJUSTLOAD(Nj)
6: ADJUSTLOAD(Ni)
7: **else**
8: Find the least-loaded node Nk.
9: **if***L* (Nk) _ *Tm* ? 2**then***fDo REORDERg*
10: Transfer all data from Nk to N = Nk_1.
11: Transfer data from Ni to Nk, s.t. L(Ni) = dx=2e and L(Nk) = bx=2c.
12: ADJUSTLOAD (N)
13: fRename nodes appropriately after REORDER.g
14: **end if**
15: **end if**

Example1: In the setting above, the maximum load is at most log log n= log d+O with high probability. Our proof (not included for reasons of space) uses the layered induction technique from the seminal work of Because of the variance in the arc length associated with each peer; we must modify the proof to take this into account. The standard layered induction uses the fact that if there is k bins that have load at least k,

Examole2: long distance links are constructed using the harmonic distribution on node-link distance. Value Link denotes the overlay when the harmonic distribution on value distance. Given the capacities of nodes (denoted by {β1, β2, · · · , βn}), we enhance the basic algorithm in Section III-B2 as follows: each node i approximates the ideal number of file chunks that it needs to host in a load balanced state as follows:

$$A_i = \gamma\beta_i,$$

Note that the performance of the Value Link overlay is representative of the performance of a plain DHT under the absence of hashing and in the presence of load balancing algorithms which preserve value contiguity.

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):

// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

V. Distributed File System

We have given several provably efficient loadbalancing for distributed file's protocols for distributed data storage in P2P systems. More details and analysis can be found in a thesis. Our algorithms are simple, and easy to implement in. distributed files so an obvious next research step should be a practical evaluation of these schemes. In addition, several concrete open problems follow from our work.

First, it might be possible to further improve the consistent hashing scheme as discussed at the end of our range search data structure. Distributed does not easily generalize to more than one order. For example (Fig.4) when storing music files, one might want to index them by both artist and song title, allowing lookups according to two orderings. Since our protocol rearranges the items according to the ordering, doing this for two orderings at the same time seems difficult. A simple, but inelegant, solution is to rearrange not the items themselves, but just store pointers to them on the nodes. This requires far less storage, and Network Setting.

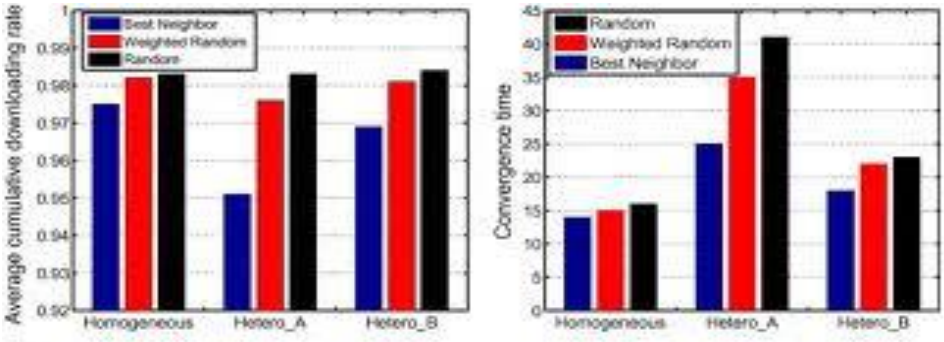

Fig.4 The average downloading rate and Convergence time

Makes it possible to maintain two or more orderings at once. Lastly, emitting nodes to choose arbitrary addresses in our item balancing protocol for distributed file's makes it easier for malicious nodes to disrupt the operation of the P2P network. It would be interesting to find counter-measures for this problem.

VI. Performance Evaluation

We run a varying number of players. The players move through the world according to a random waypoint model, with a motion time chosen uniformly at random from seconds, a destination chosen uniformly at random, and a speed chosen uniformly at random from (0, 360) pixels per second. The size of the game world is scaled according to the number of players. The dimensions are 640n _ 480n, where n is the number of players. All results are based on the average of 3 Experiments, with each experiment lasting 60 seconds. The

experiments include the bent of log n sized LRU cache long pointers. The HDFS load balancer and our proposal. Our proposal clearly outperforms the HDFS load balancer. When the name node is heavily loaded (i.e., small M's), our proposal remarkably performs better than the

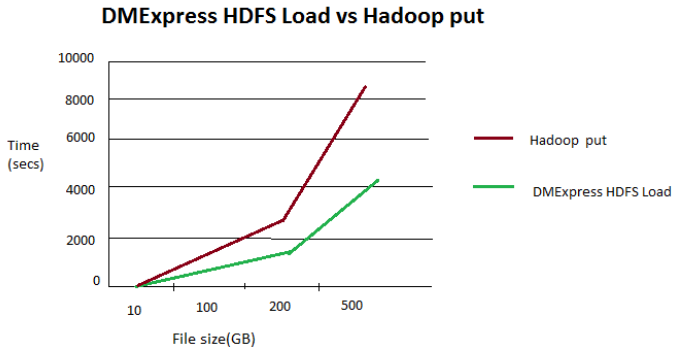

Fig.5 HDFS

HDFS load balancer. For example, if M = 1%, the HDFS load balancer takes approximately 60 minutes to balance the loads of data nodes. By contrast, our proposal takes nearly 20 minutes in the case ofM= 1%. Specifically, unlike the HDFS load balancer, our proposal is independent of the load of the name node. In particular, approximating the unlimited scenario is expensive, and the use of blog2 nc virtual peers as proposed in introduces a large amount of topology maintenance track but does not provide a very close approximation. Finally, we observe that while we are illustrating the most powerful instantiation of virtual peers, we are comparing it to the weakest choice model further improvements are available to us just by increasing d to 4.

VII. Conclusions

A 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 work is 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 (i.e., the distributions of file chunks in a large-scale storage system) 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. The computer simulation results are encouraging, indicating that our proposed algorithm performs very well.

Reference

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

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," LNCS 2218, (Nov. 2001), 161–172.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in Proc. 21st ACM Symp. Operating Systems Principles (SOSP'07), (Oct. 2007), 205–220.

[4] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," in Proc. 2nd Int'l Workshop Peerto-Peer Systems (IPTPS'02), (Feb. 2003), 68–79.

[5] D. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," in Proc. 16th ACM Symp. Parallel Algorithms and Architectures (SPAA'04), (June 2004), 36–43.

[6] D. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma -a high performance dataflow database. In Proc. VLDB, 1986.

[7] H. Feelifl, M. Kitsuregawa, and B. C. Ooi. A fast convergence technique for online heat-balancing of btree indexed database over shared-nothing parallel systems. In Proc. DEXA, 2000.

[8] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to p2p systems. Technical Report http://dbpubs.stanford.edu/pubs/2004-18, Stanford U., 2004.

[9] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In WebDB, 2004.

[10] D. R. Karger and M. Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. In Proc. IPTPS, 2004.