ISSN: 2277-9655
Impact Factor: 1.852

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Efficient Load Balancing With Distributed Hash Tables in Cloud

M.Manimegalai[*1], Ms. A.Kannammal[2]
[*1]PG Scholar, [2]Associate professor, Department of Computer Science and Engineering, Jayam College of Engineering and Technology, Dharmapuri,Tamilnadu, India
manimegalaicse@gmail.com

Abstract

Distributed file systems are key building blocks for cloud computing applications based on the Map Reduce programming paradigm. In such file systems, nodes all at once 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 on the nodes. However, in a cloud computing environment, failure is the norm, and nodes may be improved, replaced, and added in the system. This dependence is clearly incompetent in a large-scale, failure-prone environment on account of the central load balancer is put under considerable workload that is linearly scaled with the system size, and may thus become the attainment bottleneck and the single point of failure. Here, 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 outputs indicate that proposal is comparable with the existing centralized approach and considerably outperforms the prior distributed algorithm in terms of load imbalance factor, migration cost, and algorithmic overhead.

Keywords: DHT, Centralise System, LoadImBalancing, Distributed System

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 a few efforts 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 these approaches: 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 arbitrary mapping of items means that only a limited number of items land in the (small) ring interval owned by any node. The cloud computing updated 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 abundant and random, some machines receive more than their fair share, by almost a factor of n times the average. To manage with this problem, many DHTs use virtual nodes each real machine pretends to be several distinct machines, each participating individually in the DHT protocol. The machine’s load is thus driven by summing over several virtual nodes, creating a tight gathering of load near the average. As an example, the Chord DHTs 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 movement to one of them if the distribution of load in the system has changed. Since only one virtual node is alive, the real node need not pay the original Chord protocol’s multiplicative increase in space and band width costs. The solutions is therefore allows nodes to move to arbitrary addresses; with these freedom shows that load balance an arbitrary distributing the items, without exhaust much cost in maintaining the load balance. Here, the scheme works through a kind of “work stealing” in which under loaded nodes migrate to portions of the address space occupied by too abundant items. The protocol is simple and experimental, with all the complexity in its performance analysis. Here, primarily interested in studying the load rebalancing problem in distributed file systems specialized for large-scale, dynamic and data-intensive clouds. Finally, 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 identify counter-measures for this problem.

ISSN: 2277-9655
Impact Factor: 1.852

Related Work

The attempt to load-balance can fail in two ways. First, the typical “random” partition of the address space among nodes is not entirely 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 inhibit the randomization of data items addresses. For example, to support range seeking in a database application the items may need to be placed in a specific order, or at specific addresses, on the ring. In such cases, find the items unevenly distributed in address space, means that balancing the address space among nodes is not adequate to balance the distribution of items among nodes. Here the given 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 system stores a relation over a large and dynamic set of nodes, and support queries over these relations. Many current systems, known as Distributed Hash Tables (DHTs) use *hash partitioning* to ensure storage balance, and support point queries over the relations. There has been considerable recent interest in developing P2P systems that can support efficient ranges of query. 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 familiar that hash partitioning is inefficient for answering such *ad hoc* range of 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, applying the NBRADJUST operation. Behind this split, there may be NBRBALANCE violations between two pairs of neighbours and in responses, 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 governs data for a particular range. When the node leaves, the data is stored becomes unavailable to the rest of the peers. P2P networks resolve 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 various 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 lists. It supports logarithmic time range-based lookups and guarantees path locality. Mercury is more general than Skip Net because it supports range-based lookups on multiple-attributes. Our use of random sampling to evaluate query selectivity constitutes a novel contribution towards implementing scalable multi-dimensional range queries. Load balancing is another essential way in which Mercury from Skip Net. While Skip Net integrates a constrained load-balancing mechanism, it is only appropriate when part of a data name is hashed, in which case the part is unavailable for performing a range query. This shows that Skip Net supports load-balancing or range queries not both.

System Model

(a)Data Popularity:

Unfortunately, in many applications, a particular range of values may show a much greater popularity in terms of database insertions or queries than other ranges. This would cause the node accountable for the popular range to become overloaded. One obvious solution is to conclude some way to partitioning the ranges in proportion to their popularity. As a load pattern varies, 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 decide the average load existing in the system, and, hence, can resolve if it is relatively heavily or lightly loaded. Second, the histograms consist of information, about which parts of the overlay are lightly loaded.

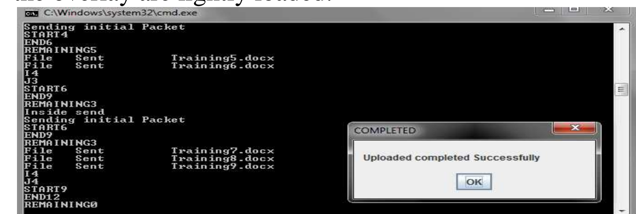

Fig.1 about which parts of the overlay are lightly loaded

(b)Load Balancing:

We have shown how to balance the address space. Some applications, such as those designing to support range-searching operations, need to designate an appropriate, non-random mapping of items into the address space. In this section, we check a dynamic

ISSN: 2277-9655
Impact Factor: 1.852

protocol that aims to balance load for *arbitrary* item distributions. To do so, we must forfeit the previous protocol's restriction of each node to a small number of virtual node locations—instead, each node is free to drift 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 ensuing, 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.

(c)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 allow nodes to self-organize and Repair while constantly offering lookup functionality in node dynamism, clarifying the system provision and management. The chunk servers in our proposal are arranged as a DHT network. Typical DHTs assurance that if a node leaves, then its locally hosted chunks are accurately migrated to its successor; if a node joins, then it allocates the chunks whose IDs promptly precede the joining node from its successor to manage. Now we explain the application of this idea to DHTs. Let h0 be a universally admit hash function that maps the peers onto the ring. Correspondingly, let h1; h2; : : :hd be a series of universally agreed hash functions mapping the items onto the ring. To embed 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 values

(d) ***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 normally proportional to the number of file chunks the node obtains. Because the files in a cloud can be promptly created, deleted, and appended, and nodes can be upgraded, recovered and added in the file system, the file chunks are not distributed as similarly as possible among the nodes. Our objective is to assign the chunks of files as uniformly as possible among the nodes such that no node manages an excessive number of chunks.

(e)Replica Management:

In distributed file systems (e.g., Google GFS and Hadoop HDFS), a constant number of replicas for every file chunk are maintained in distinct nodes to improve file availability with respect to node failures and takeoff. Our current load balancing algorithm does not treat replicas clearly. 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 particularly, each under loaded node samples a number of nodes, each preferred with a probability of 1/n, to share their loads (where n is the total number of storage nodes).

Load Balancing Algorithm

In our proposed algorithm, each chunk server node I first estimated whether it is under loaded (light) or overloaded (heavy) without global awareness. A node is *light* if the number of chunks it hosts is smaller than the threshold. Load statuses of a sample of randomly preferred nodes. Fig.2 shows that the concept of Load balancing

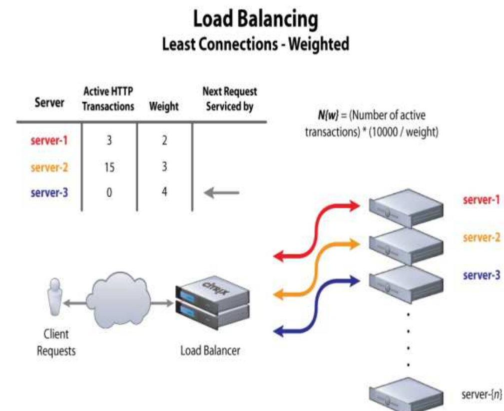

Fig.2 Load Balancing

Specifically, each node contacts a number of randomly selected nodes in the system and builds a vector denoted by V. A vector abide of entries, and each entry includes the ID, network address and load status of a randomly preferred node. Fig. 3 shows the total number of messages generated by a load rebalancing algorithm, A large-scale distributed file system is in a

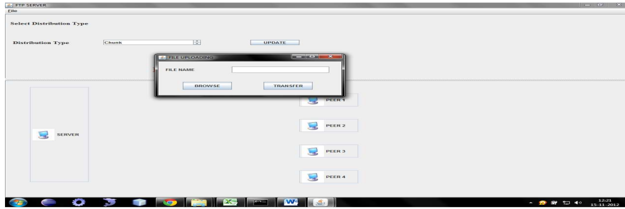

Fig.3 Load Balanced State

Load-balanced state:

if each chunk server hosts no more than Am chunks. In our proposed Load-balanced 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 free 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 present least-loaded node, say node i abandon 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 performed to release the extra load on the next heaviest node in the system. This process repeated 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. Particularly; we discuss the tradeoff between the value of nV and the movement cost. A larger nV proposes more overhead for message transfers, but results in a smaller movement cost.

Procedure 1 ADJUSTLOAD (Node Ni) fOn 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 for 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, where 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 maximal 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 using the fact that if there is k bins that have load at least k,

Example2: long distance links are constructed using the harmonic distribution on node-link distance. Value Link means 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:

$$A_i = \gamma\beta_i,$$

Notes 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));
```

Distributed File System

We have given several provably efficient load balancing for distributed file's protocols for distributed data storage in P2P systems. Further details and analysis can be found in a thesis. Our algorithms are simple, and effortless to implement in. distributed files so clearly next research step should be a practical evaluation of these schemes. In addition, several detailed open problems follow from our work. First, it might be possible to further enhance the consistent hashing scheme as discussed at the end of our range search data structure. Distributed does not conveniently generalize to more than one order. For example (Fig.4) when storing the music files, one might want to index them by both artist and song title, granting lookups according to two orderings. Since our protocol readjusting the items using the ordering, performing this for two orderings at the

same time seems difficult. A simple, but insignificant, solution is to rearrange not the items themselves, however just store pointers to them on the nodes. This needs far less storage, and

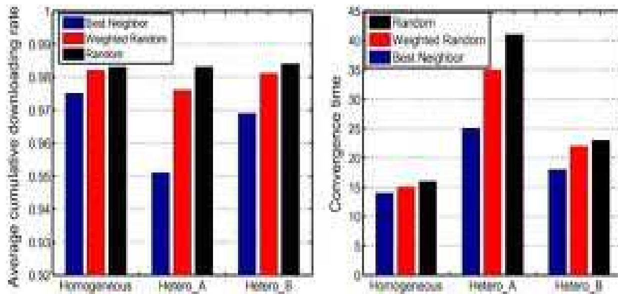

Fig.4 The average downloading rate and Convergence time in Network Setting

Makes it possible to maintain two or more orderings at once. Lastly, permitting 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 engaging to find counter-measures for this problem.

Performance Evaluation

We run a varying number of players. The players manoeuvre through the world according to a random waypoint model, with a motion time selected uniformly at random from seconds, a destination selected uniformly at random, and a speed selected uniformly at random from (0, 360) pixels per second. The size of the game world is scaled according to the number of players. The range of dimensions are 640n _ 480n, where n is the number of players. All outcomes are based on the average of 3 Experiments, with each experiment persisting for 60 seconds. The experiments include the bent of log n sized LRU cache long pointers. The HDFS load balancer and our proposed idea. Our proposal clearly concludes the HDFS load balancer. When the name node is heavily loaded (i.e., small M's), our proposal remarkably performs better than the

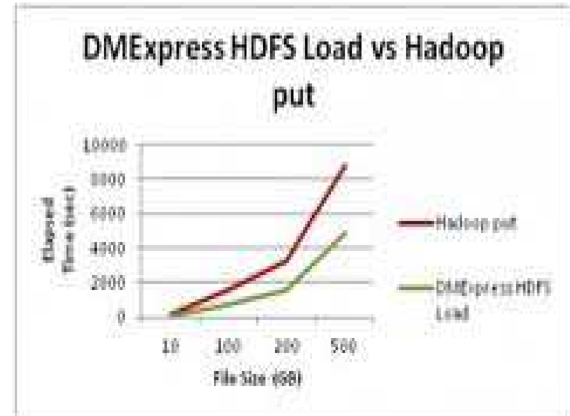

Fig.5 HDFS

HDFS load balancer. For example, if M = 1%, the HDFS load balancer takes approximately 60 minutes to balance the loads of data nodes. By comparison, our proposal takes nearly 20 minutes in the case of M= 1%. Specifically, unlike the HDFS load balancer, our proposed idea is independent of the load of the name node. Especially, approximating the unlimited scenario is costly, 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 notice that while we are illustrating the most powerful instantiation of virtual peers, we are correlating it to the weakest choice model further improvements are available to us just by increasing d to 4.

Conclusions

A novel load balancing algorithm to deal with the load rebalancing problem in large-scale, dynamic, active and distributed file systems in clouds has been presented in this paper. Our proposal aims to balance the loads of nodes and reduce the demanded movement cost as much as possible, while taking benefits of physical network locality and node heterogeneity. In the absence of typical real workloads (i.e., the distributions of file chunks in a large-scale storage system) in the public domain, we have tested the performance of our proposal and compared it against competing algorithms through synthesized probabilistic distributions of file chunks. The combination workloads stress test the load balancing algorithms by creating a few storage nodes that are heavily loaded. The computer simulation outcomes are encouraging, indicating that our proposed algorithm works very well.

Reference

[1] 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, pp. 17–21, Feb. 2003.

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

[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, pp. 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 Peer to- Peer Systems (IPTPS'02)*, Feb. 2003, pp. 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, pp. 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] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[8] 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.

[9] 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.

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

[11] S. Ghandeharizadeh and D. J. DeWitt. A performance analysis of alternative multi-attribute declustering strategies. In *Proc. SIGMOD*, 1992.

[12] N. J. A. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. USITS*, 2003.

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