A Dynamic Popularity-Aware Load Balancing Algorithm for Structured P2P Systems

Narjes Soltani, Ehsan Mousavi Khaneghah,
Mohsen Sharifi, and Seyedeh Leili Mirtaheri

School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
{emousavi,mirtaheri,msharifi}@iust.ac.ir,
narjes_soltani@comp.iust.ac.ir

Abstract. Load balancing is one of the main challenges of structured P2P systems that use distributed hash tables (DHT) to map data items (objects) onto the nodes of the system. In a typical P2P system with N nodes, the use of random hash functions for distributing keys among peer nodes can lead to O(log N) imbalance. Most existing load balancing algorithms for structured P2Psystems are not proximity-aware, assume uniform distribution of objects in the system and often ignore node heterogeneity. In this paper we propose a load balancing algorithm that considers node heterogeneity, changes in object popularities, and link latencies between nodes. It also considers the load transfer time as an important factor in calculating the cost of load balancing. We present the algorithm using node movement and replication mechanisms. We also show via simulation how well the algorithm performs under different loads in a typical structured P2P system.

Keywords: Structured P2P Systems, Load Balancing, Node Movement, Replication.

1 Introduction

In most structured Peer-to-Peer (P2P) systems distribution of objects among nodes is done through distributed hash tables (DHT) mechanisms that use consistent hashing to map objects onto nodes [1]. Using this mechanism, a unique identifier is associated with each data item (object) and each node in the system. The identifier space is partitioned among the nodes that form the P2P system and each node is responsible for storing all data items that are mapped to an identifier in its portion of the space.

If node identifiers are chosen at random (as in [1]), a random choice of item IDs result in O(log N) imbalance factor in the number of items stored at a node. Here N is the total number of nodes in the system. Furthermore imbalance may result due to non-uniform distribution of objects in the identifier space and a high degree of heterogeneity in object loads and node capacities, memories, and bandwidths.

J.J. Park et al. (Eds.): NPC 2012, LNCS 7513, pp. 77–84, 2012.
© IFIP International Federation for Information Processing 2012

Several solutions are offered to solve the load balancing problem like the one proposed in [2], but these solutions usually have some shortcomings. For instance lots of them do not consider system dynamicity, nodes and objects heterogeneity, link latency between nodes, and the popularity level of moved items. Our algorithm uses two mechanisms namely nodes moving and replication to balance the load between nodes with consideration of items popularities.

The rest of the paper is organized as follows. In Section 2, we formulate the load balancing problem more explicitly and in Section 3, we discuss the related work. In Section 4, we describe our algorithm and we evaluate it in Section 5. We discuss conclusion in Section 6.

2 Definitions and Problem Formulation

We define node load as the temporal average number of bytes which is transferred by that node in each unit of time. In the same way we define the node capacity as the maximum number of bytes that node can transfer per time unit. A node is overloaded if its load is more than an upper threshold which is defined relevant to node capacity.

Our load balancing algorithm aims to minimize the load imbalance factor in the system while also minimizing the moved load. Calculation of load destination cost in our algorithm is based on its load and uptime and also its proximity to the overloaded node. Since there is no global information in P2P systems, we do not claim to select the best node in the system to move load to, but our algorithm does this in a group of nodes. Later we explain how these groups are formed. So when we want to move some load from a node *i* to a node *j* the destination cost is formulated as below:

$$DestinationCost = w_1 * load_j / cap_j + w_2 * (loc_i — loc_j) / distance_{max} + w_3 * (uptime_j / t) \quad (1)$$

In (1) *cap* and *loc* denote the capacity and location of a node in order. To normalize the location parameter in the above formula, we divide subtract of locations by $distance_{max}$ which stands for the distance between *i* and the farthest node *j* in the mentioned group. In the above formula *t* is the period of time our experiment lasts and we divide uptime of node *j* by it to normalize the uptime parameter. Also w_i $(1<=i<=3)$ is the weight given to different cost function parameters and $\sum w_i=1$ is always held. These weights are application-defined.

The load of each node *j* is defined as the summation of its data items loads. The load of each object *k* is defined as follows:

$$Load_k = size_k * access_frequency_k \quad (2)$$

In the above formula we want to calculate the average amount of bytes that is transferred in each unit of time in relation with object *k*. We calculate frequency access periodically and in distinct intervals. This means that we consider *r* intervals each last for *t'* seconds, then divide the number of access in each interval by the subtract result of $r*t'$ and t_{first}. Starting intervals times from zero in each round of simulation, t_{first} stands for the beginning time of each interval namely it is set to 0, t', $2t'$, … and $(r-1)t'$ in order for each of the *r* intervals. These intervals can be stored in a cyclic queue with limited size in which the old intervals are replaced with the new ones in a

cyclic manner. Next we sum the achieved values to calculate the *access_frequency* parameter.

$$access_frequency = \sum_{p=0}^{r-1} objAccessNo_{p+1} / (r * t' - p * t') \quad (3)$$

Defining a node n utility as $load_n/capacity_n$, the goal of our algorithm is to close nodes' utilities to each other as much as possible. However it is not always the case because our algorithm balances the load with considering its cost.

3 Related Work

Generally load balancing protocols are divided to two main groups in structured P2P systems. The first group is based on uniform distribution of items in identifier space and the second group has no such assumption [3]. Suppose that there are N nodes in the system, load balancing is achieved in the first group if the fraction of address space covered by each node is O(1/N). Most of algorithms have used the notion of virtual servers, first introduced in [1] to achieve this goal. A virtual server is similar to a single pear to the underlying DHT and has its own routing table and successors list.

Chord suggests each physical node hosts O(log N) virtual servers which leads to each node has some constant number of items with high probability [1]. CFS [4] accounts for nodes heterogeneity by allocating to each node some number of virtual servers proportional to the node capacity. In [2] Rao et.al. have proposed three different mechanisms to balance the load using virtual servers, yet their mechanisms are static and ignore data items popularities.

Using virtual servers in any algorithm leads to some common disadvantages. The first is that it leads to churn increase. Another disadvantage about virtual servers is that using them causes a great increase in routing table entries. Considering the above problems about virtual servers, our algorithm does not use of virtual servers [5].

Protocols which do not assume uniform item distribution use two different mechanisms to achieve load balance, namely item movement [3] and node movement [6]. Moving items break the DHT assignment of items to nodes, so that items cannot easily be found any more. Moving nodes by letting them to choose their own addresses arbitrarily increase the threat of Byzantine attack which can prevent some items from being found in the network.

Replicating data items is another way to achieve load balance. Although, some simple replication mechanisms have been proposed in structured P2P systems like chord [1], but none of them does this dynamically and with consideration of variant system loads.

4 Load Balancing Scheme

4.1 Nodes Load Information

When a node wants to join the system, a unique key is given to it using a hash function we call *"First Hash"*. For the purpose of load balancing a set of load directories,

each called *LoadDir* is designed in the system to which nodes send their loads, capacities, locations, and uptimes information periodically. A node's uptime is defined as the average of continuous time it stays in the system.

To prevent Byzantine attack we use the proposed way in [6]. Each node connects to a central authority once, i.e. the first time it joins the system and obtains a directory identifier, we call *IDdir* that specifies to which *LoadDir* the node should send its information. The number of distinct directory identifiers is limited and determines the number of load directories in the system.

Grouping of nodes in our load balancing algorithm is done based on their directory identifiers. This means the nodes with the same value of *IDdir* send their information to the same *LoadDir* and in case of overloading, our algorithm first tries to move load between the nodes in the same group. A directory with the identifier *d* is stored in the first node whose identifier is equal to or follows *d* and when this node wants to leave system it has to send its stored load directory to its successor.

The central authority periodically sends the directory identifiers to the related nodes, so that each directory is aware of other directories.

4.2 Load Balancing Algorithm

A node starts load balancing algorithm when it notices its load more than its upper threshold. In the simulations it is proved that setting each node upper threshold to 95% of its capacity generates the best result.

Every node checks its load periodically and in case of overloading; it puts its popular items in a list called *popular-item-list.* This list is stored separately in each node and in relation with its own items. In our algorithm an item is popular if more than a quarter of the node load is due to that item load.

Our algorithm uses two mechanisms namely nodes moving and replication to balance the load between nodes with consideration of items popularities. In the following parts we explain these mechanisms in detail.

Node Movement

Node movement is done when one of the following cases arise:

1. A node gets overloaded due to the high popularity of more than one of its items.
2. A node gets overloaded because of high amount of data items put on it while none of them is highly popular.

Considering the above conditions, if a node *n* is overloaded, it sends a request to its relevant directory asking it to find a proper underloaded node for moving load. The selected under-loaded node should leave its previous location in the overlay and join at a new location specified by the *n*'s directory called *"split point"*. The overloaded node's directory selects some of *n*'s data items, starting from the item whose key has the most distance from node key and checks whether *n* load reaches to a normal level by moving this data item or not. If so, the split point is set to that data item key; Otherwise selection of data items continues in the same order until *n* load gets normal or

the only remained data item be the one whose key is equal to overloaded node key, in this case the only remained data item is of course a very popular data item and by iterative execution of our algorithm, the next time this node is an eligible candidate for the second load balancing mechanism and its load is balanced by that way.

Finally the split point is set to the last selected item's key. In simulation we show setting normal load of any node to 75% of its capacity is an appropriate choice.

Selection of a proper under-loaded node is done in two steps and as follows. In the first step the overloaded node's directory searches in its stored information, calculates the destination cost function stated in section 2 for each of its entries and selects the one with the minimum cost. The selected node *m* should move to the specified split point, so all of its assigned keys should be reassigned to its successor. In the second step the directory checks two conditions. The first is that the reassignment process does not lead to the *m*'s successor overloading and the second is moving selected items from *n* to *m* do not end up with *m*'s overloading. If both of these conditions are held, *m* is the proper node we are searching for and it should leave and rejoin to system at the specified split point. If any of the conditions is not held, the selection process is repeated from the first step. If necessary, this directory can connect to other directories and selects a proper node from them.

Replication

If a node is overloaded because of the high popularity of one of its items, it is probable that due to its increasing popularity rate, moving this item to another node causes that node to get overloaded too and it is better to replicate it.

For the purpose of replication, we use a second hash function called *SecHash* and also a set of replication directories each called *RepDir*. Each entry of these directories consists of two parts, namely the name of replicated data item and the destination of replication. Creation of replication directories is done dynamically throughout system operation. If a node wants to replicate one of its items named *A*, it should search for successor of *SecHash(A)* to find the *RepDir* where it should add an entry.

The replication destination is specified by the overloaded node's *LoadDir*. This directory calculates the destination cost function stated in section 2 for each of its entries and at last the one with the minimum cost is selected if by moving half of the *A*'s load to it, it does not change to an overloaded node. If this condition is not held, the next minimum cost nodes are tested until an appropriate node is selected. Again this directory can connect to other directories if necessary. *A* is replicated in the found node in association with another field where *FirstHash(A)* is stored. This field is used during search process as we explain later.

The overloaded node can then refuse the replicated item's received requests until it reaches to a normal load state. By this way if a node is searching for a replicated item, it may receive no result after a period of time which means a timeout has occurred. If this is the case, the requester node uses the second hash function to find the relevant *RepDir* and reads the replication destination from it. The above process can be extended to include replicating on more than one node if necessary.

4.3 Search Mechanism

Every time a node receives a request with key k, it checks whether *k* falls in the interval it is responsible for or not. If it is the case, this node returns the requested item. Otherwise the node should forward the request to another node with respect to its finger table [1], but in our algorithm this step delays with another step in which the node checks whether a replica of the searched item is stored in itself. Since intermediate nodes have no information about the name of requested items during search process and work only with hashed keys, the node checks the requested item key with the fields associated to replicated items on it. If no match is found, it forwards the request to another node with respect to its finger table, otherwise it response to the requester node.

To make the system fault tolerant, we can backup replication directories in the l next successors of the nodes where they are stored. The value of l is defined with consideration of the fault tolerance level needed in system.

5 Simulation Results

To evaluate our algorithm, we have designed and implemented a simulator in java based on Chord structured P2P system. Throughout the simulation we have shown the way our algorithm balances the load and also the importance of proximity factor in bandwidth consumption.

Our simulated nodes are completely heterogeneous and with different capabilities, so Pareto node capacity distribution with parameters shape=2 and scale=100 is used. In our simulated environment, nods can leave or join the system at any time. As stated before our algorithm aims to close nodes' utilities to each other as much as possible. It is not however always the case, because our load balancing algorithm considers the imposed cost.

Fig.1(a) shows nodes utilities in the system before applying any load balancing algorithm. It illustrates the large difference between nodes utilities. In this situation, lots of nodes are overloaded while there are also a lot of nodes with very low or even zero load. Applying our load balancing algorithm, the results change as in Fig.1(b).

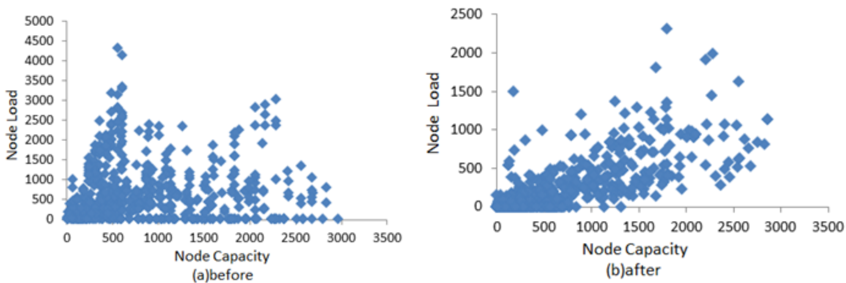

Fig. 1. Load balancing performance (a)Before load balancing (b)After load balancing

To demonstrate the importance of proximity in our algorithm, Fig.2 displays the bandwidth consumption of load transfer through the hops passed during load balancing process. As it is shown, when we have relaxed the proximity factor, the passed physical hops is much more than the case we have regarded this factor.

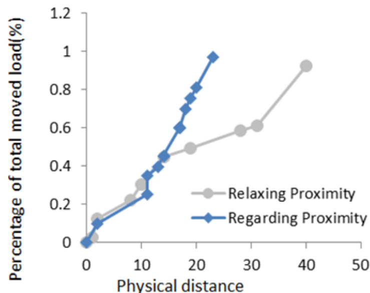

Fig. 2. Changing of links delay while load balancing

In Fig.3 we show the effectiveness of our algorithm by comparing it with three other algorithms, namely Rao et.al. algorithm [2], CFS algorithm [4] and log(N) virtual Server algorithm[1]. As we have mentioned previously Rao et.al. proposed three different schemes to balance the load using virtual servers. In this section we compare our algorithm with the "one-to-one" scheme in which one node contacts a single other node per unit time as two other schemes are said to utilize nodes similarly.

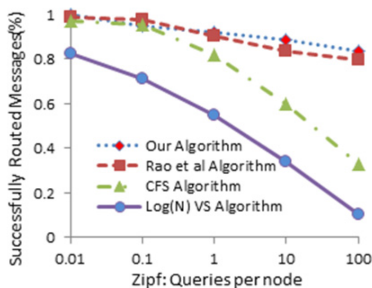

Fig. 3. Percent of successfully routed queries for trace-driven simulation with varying load

The focus was put on the percentage of successfully routed queries for trace-driven simulations with varying loads. To this end we have used Zipf query distribution. This experiment examines how the load balancing algorithms responded to different degrees of applied workload. In almost all cases, we found our algorithm performs the same as or better than the other algorithms.

6 Conclusion

This paper presents a load balancing algorithm which considers items non-uniform distribution, heterogeneous nodes, system dynamicity, and objects different and

variable popularities. Also two important factors namely the uptime and proximity are considered during load transfer process. For the purpose of load balancing we have used different mechanisms including moving node and replication. Simulation results show that running our algorithm causes node's utilities to close to each other in most cases.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications, New York, NY, pp. 149–160 (2001)
2. Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load Balancing in Structured P2P Systems. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS 2003. LNCS, vol. 2735, pp. 68–79. Springer, Heidelberg (2003)
3. Ruhl, J.M.: Efficient algorithms for new computational models, USA, Techreport (2003)
4. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. SIGOPS Oper. Syst. Rev. 35(5), 202–215 (2001)
5. Sharifi, M., Mirtaheri, S.L., Mousavi Khaneghah, E.: A Dynamic Framework for Integrated Management of All Types of Resources in P2P Systems. The Journal of Supercomputing 52(2), 149–170 (2010)
6. Rieche, S., Petrak, L., Wehrle, K.: A thermal-dissipation-based approach for balancing data load in distributed hash tables. In: Proc. of 29th Annual IEEE Conference on Local Computer Networks (LCN), Germany, pp. 15–23 (2004)
7. Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. In: ACM SIGOPS Operating Systems Review, OSDI 2002: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, New York, NY, USA, pp. 299–314 (2002)