An Efficient Method for Handling Data with DHT and Load Balancing Over P2P Network

Satheesh B R
Department of Computer Science Engineering

Abstract— Peer to peer systems is a class of decentralized and distributed system. The participating node in the peer to peer systems acts as a both client and server whenever it is needed for another peer system. Whenever the storage is done on a system it is shared among all peers and also whenever the retrieval is done it is also shared among all peers. This makes an advantage of reliability, robustness and scalability. To avoid a situation of getting heavier loads than other peers the peer to peer systems must be provided with the loads to the all participating peers. This covers sending and receiving message that is the communication cost and computational power for requesting process. When the messages an forwarded from one peer to another peers they are exposed to a routing loads for queries during the information lookup and they only traverse them. And the traffic loads usually consist of both communication and computational power. The loads balancing in the peer to peer system are mainly based upon DHTs. The DHT are generally made to be well balanced under a flow of request and the request must be uniform. For example the objects having equal popularity that is the similar amount of request should be sent to all nodes

Keyword: - Distributed systems, DHT, Peer – to – Peer Systems, Routing Loads, Traffic Load Balancing.

I. INTRODUCTION

Load balancing in the hash table shares common challenges in some respect solution with other domains such as network load balancing [1]. The network load balancing is used as a multiple inter faces that is used for simultaneous data transmission and multiprocessor programs which is scheduled and the processor must be assigned to a tasks to obtained the lowest completion time. DHT requires load balancing algorithm as because of their decentralized structure for specific properties.

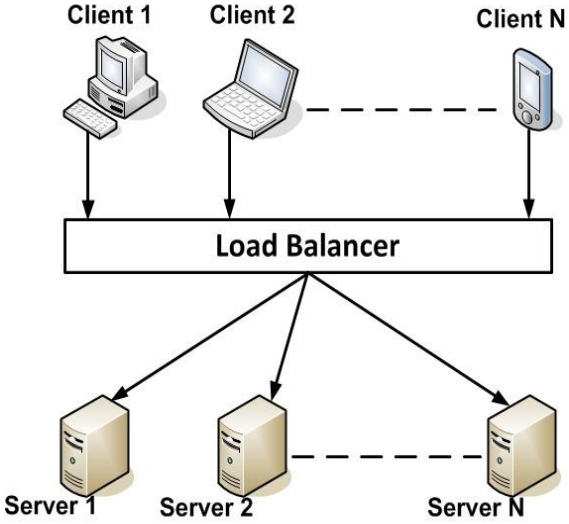

Fig. 1: Framework for P2P load balancing system

II. LITERATURE SURVEY WORK

This network is majorly composed of set of nodes with the addresses basically called identifiers given within an identifier space .to have a connected structured the pair of nodes are joined together by the links. The nodes collaboratively store a large number of objects, with each peer responsible for a small fraction of them in a overlay structure. The identifier space is divided into a set of non-overlapping ranges that are assigned to the individual nodes. Each object can also have an identifier that is laid under a same identifier space as the nodes. The objects are stored under the responsibility of the nodes that owns the identifier range to which the object belongs[2]

When the node is giving a query (lookup query) to retrieve an object the routing functions redirects the query to the node responsible for this object. After this, the redirected request passes through multiple nodes on its way depending upon the links connected between the peers in the overlay and the lookup algorithm. Routing is basically based upon simple greed's algorithm which is operated by the means of proximity and distance functions defined on a identifier space. Usually a node greedily forwards package to the neighbour that is closest in the identifier space.

The objects are uniformly shared among the nodes when the nodes and keys are uniformly distributed over identifier space. To achieve this property one of the classical approach is namespace balancing.

A. Routing tables

Routing table consist of a list of outgoing links leading to its immediate neighbors. In a typical DHT process each node maintains its routing table. On a same time every peer contains a set of incoming links from the neighboring hood but it may not know the identity of the neighbors nor the number of links if in case they are unidirectional. Whenever routing algorithm is executed aim the forwarded node along the lookup path it will select as next hop (i.e.) neighbor from its routing table than is closer to the destination. A links with the smaller number of incoming links receives fewer requests than with the many incoming links. If the traffic loads is to be in a good way the routing tables must be organized in a way that number of incoming links per node is balanced.

We should keep updating the routing tables as a part of DHTs maintenance mechanism [2]-[3]. For that we should periodically conform whether the neighbors are still reachable and if not an obsolete entry is replaced by one of the node that conforms to the organization rules of routing tables.

In a DHT process we can improve the load balancing by considering multiple redundant path towards destination and choose a least loaded among the possible alternatives. Moreover if the routing algorithm uses same the same destination for many entries there is a chance of increasing a point of failure in a complete process.

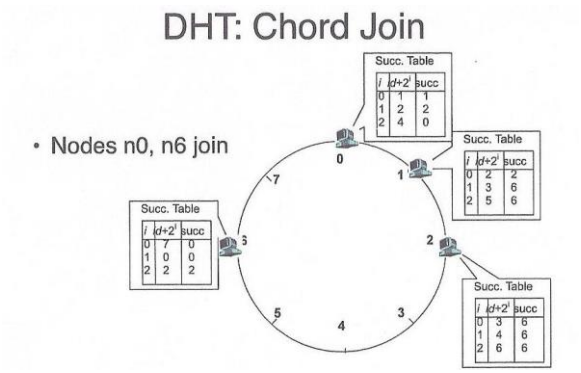

Fig. 2: DHT chord join

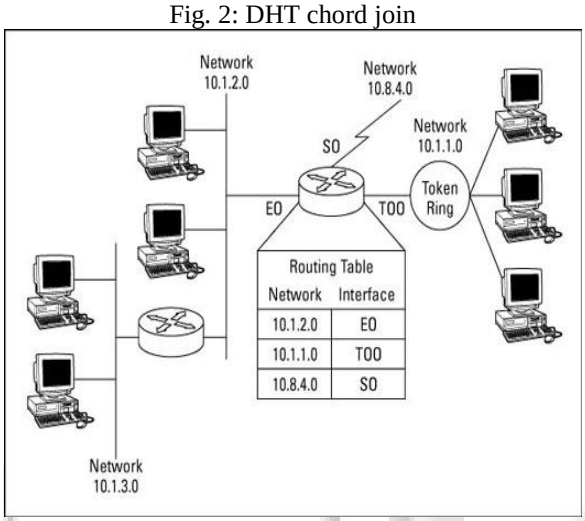

Fig. 3: Routing tables

B. Underlying topology

The communication in the underlying network structure is essentially important. Viewing a situation in which immediate neighbour in a overlay are placed in a distinct region of the underlay. At this situation the queries that are coming to destination may go back and forth in a underlay which will consistently increased the traffic in a underlay process. To properly account for this possible cause of overhead network friendly peer to peer systems restrict the identifier assignment are the rooting table organisation by using network layers metrics like delay, hop-count etc[7].

In a node every node and key arranged in m-bit identifier in a ring-shaped space that contains 2^m difference addresses. The identifier is taken by hashing a nodal IP address or a file name[7][8]. In the first node it follows an object key in a clockwise direction along the ring is mostly responsible for this object. For routing property every nodes contains a routing table. It consist of m entries and I entries. Here the I entries points towards the first following node along the ring and it is of distance at least 2^i where I=0,……,m-1. [4].At times the entry I is also called as a finger I while the corresponding links pointing at node n are n's incoming links. Chord uses greedy routine in which the requests are in a clockwise direction that makes possible for a forwarding node to pick a closest possible neighbor as a next hop to a destination. The diameter network in a chord is defined as the number of intermediate forwarding nodes along the shortest path between the most distant and destination.

Hash tables are a data structure for storing and retrieving unordered information, whose and its primary operations are in complexity class O(1) -independent of the amount of information stored in the hash table. We saw that digital trees had this same property, but only for special keys (that were digital: meaning we could decompose them into a first part of the key, a second part of the key, etc. as we can with digits in a number and characters in a String)[4]. Hash tables work with any kind of key. The most commonly used implementations of the Set and Map collection classes in Java (which are unordered) are implemented by hash tables. Here are some terms that we need to become familiar with to understand (and talk about) hash tables: hash codes, compression function, bins/buckets, overflow-chaining, probing, load factor, and open-addressing. We will discuss each below.

III. Our Approach:

We will start by discussing linear searching (using a linked list) of a collection of names. If we instead used an array of 26 indexes and put in index 0 a linked list of all names starting with "a", and in index 1 a linked list of all names starting with "b", ... and in index 25 a linked list of all names starting with "z", we could search for a name about 26 times faster by looking just in the right index for any name (according to its first letter). This speed increase assumes each letter is equally likely to start a last name, which is not a realistic assumption.

In fact, if we used an array of 26x26 (676) indexes, storing in index 0 alinked list of all names starting with "aa", and in index 1 a linked list of all names starting with "ab", ... and in index 676 a linked list of all names starting with "zz", we could search for a name 676 times faster by looking just in the right index for any name (according to its first two letters). This speed increase assumes each letter pair is equally likely to start a last name, which is not a realistic assumption.

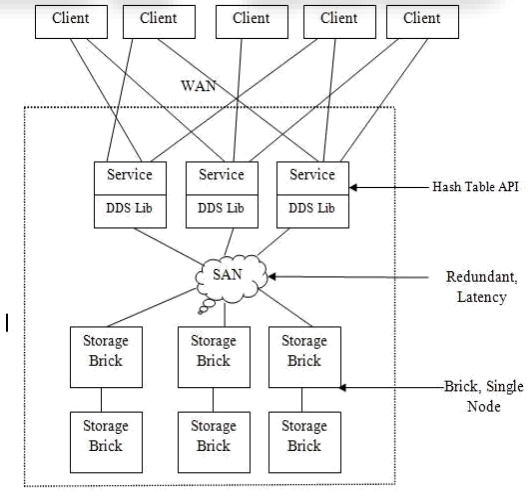

Fig. 4: Routing Table

Of course, this speedup isn't achieved unless we have at least 676 names, and each box is equally likely to have a name (which isn't true: few names start with combinations like "bb", etc). And what about looking-up information that isn't Strings: for example the Word Generator uses a List of Strings as the key for its map. So while this approach seems promising, we need to modify it to be truly useful.

A. Hash Codes

Hashing is that modification. We declare an array with any number of "bins or buckets" and use a "hash code" to compute an nit value for any piece of data that can go into the hash table. It must always compute the same hash code forth same value, so it cannot use random numbers. [5].We should design such a hash code to generate the widest variety of numbers (over the range of allintegers), with as small a probability as possible of two different values hashing to the same number.

Of course, in the case of using Strings as values, there are more Strings thanint values. There are only about 4 billion different ints -actually, exactly4,294,967,296- but an infinite number of Strings, which can be of any length, meaning any number of characters: even if we consider only Strings with lower-case letters, there are 26^N different Strings with N charters; 26^7 is8,031,810,176, so there are already more 7-letter Strings than ints. Once we have a hash code function, we use a "compression function" to convert the hash code to a legal index in our hash table. One simple compression unction computes the absolute value of the hash code (hash codes should cover both negative and positive values but array indexes are always non-negative)and then computes the remainder (using the % operator) using the hash table length as the 2nd operand, producing a number between 0 and length-1 of the hash table. Other compression functions use-bitwise operations to compute a bit pattern in the correct range.

The hash Code method must be important: it is one of the few methods declared in the Object class, so every class can override it (it is as fundamental as to String and equals, which are also declared in Object). Here is a slightlysimplified hashCode for the actual String class in Java (we will see the exactcode later).

```
public int hashCode()
{
int hash = 0;
for (int i = 0; i < chars.length; i++)
{
hash = 31*hash + chars[i];    //promotion of char -> int: its ASCII value
return hash;
}
}
```

"a".hashCode() returns 97 ('a' has an ASCII value of 97; you can actuallycall .hashCode on any String literal, which is really replaced by a Stringobject storing that value) and "aa".hashCode() returns 3104 (31*97 + 97). Generally, if String.length() is n (the chars array contains n values), thenits hashed value is given by the formula

chars[0]*31^(n-1) + chars[1]*31^(n-2) + ... + chars[n-2]*31^1 + chars[n-1]

So, "ICS23".hashCode() returns 69,494,394, and "Richard Pattis".hashCode()returns -125,886,044! Yes, because of arithmetic overflow and the standardproperties of binary numbers, the result might be negative (and overflow of negative numbers can go positive again)[9]. Recall that Java does not throw anyexceptions when arithmetic operators produce values outside of the range ofint: hashing is one of the few places where this behavior produces resultsthat are still useful.

Generally the hash Code for all the numeric types is a numeric value with that bit pattern. Characters hash to their ASCII values. Every other type in Java is built from these: for examples Strings are an ordered sequence of charsand we compute the hash code of the String by looking at every char in it. Note that "ab".hashCode() != "ba".hashCode(), which is fine because those twoStrings are not equal: the order of the letters is important. In fact, a good hash Code function for any ordered data type will produce different values for different orders of the same values.

Likewise, here is the Java hash code for a List (defined in the AbstractListclass). It relies on computing the hash code for every value in the list (and if a null value is in the list, using 0 for its hash code). Computing the hash code for a sequence of values in a list is similar to computing the hash code for a sequence of characters in a String, because the order is important.

```
public int hashCode()
{
int hashCode = 0;
for (E e : this)
{
hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
}
return hashCode;
}
```

Note that if we store an explicit null in a List (this is legal) we cannotcall the hashCode method on it: Java would throw a null pointer exception,so we use just the value 0 for its hash code.Recall in the WordGenerator program we used a key that was a list of String.So we would iterate through the list of String values, computing the hash codefor each String in the list, and combining them as shown above.

It is critical that the equals and hashCode methods for a class are compatible.The key property is that if a.equals(b) then a.hashCode() == b.hashCode().Of course the opposite is not true, because many different Strings have equalhashCodes, because there are more Strings than ints: of course it is unlikelythat many different Strings actually used in some problem will have the samehash code (if there are only millions, not billions, of them).

This compatibility requirement is very important for unordered collections likeSets. Typically we iterate through the values of a Set to compute the hashCode.But, values in the Set can be stored in (and iterated through in) any order.Regardless of the order these values are processed, they must compute the same hash code each time hashCode is called (because set implementations thathappen to store their values in different order are still .equals).So, hash codes are different, but only slightly, for unordered collections(like a Set). Since no matter what order the values are stored, the hash codesshould be the same, we cannot use the hash code method above, but instead mustuse something that accumulates the hash code values of its elements withoutregard to their order. Here we just add together (without the weighting of 31*)all the values.

```
public int hashCode()
{
int h = 0;
```



```
for (E e : this)
{
if (e != null)
h += e.hashCode();   //or we could use h *= e.hashCode()
}
return h;
}
```

Thus, if we add or multiply together all the hash codes, it doesn't make adifference what order we do the addition or multiplication: a+b+c, b+a+c,c+b+a, etc. all compute the same value (as does a*b*c, b*a*c, c*b*a, etc.)

Finally, here is something that I found interesting, when I came across it whenI was reading the .java String class). The real hashCode method in String lookslike the following, with cachedHash being an instance variable for all objectsin the String class, which is initially set to 0. The first time hashCode is called, cachedHash is 0 so it computes the hash value and stores it incachedHash[6] before returning it. For every other time it is called, itimmediately returns cachedHash, doing no further computation. Remember thatStrings are immutable, so once they are constructed their contents do notchange, so once the hashCode is computed for a String object, that String object will always return the same result.

```
public int hashCode()
{
if (cachedHash != 0)
return cachedHash;
int hash = 0;
for (int i = 0; i < chars.length; i++)
{
hash = 31*hash + chars[i];   //promotion of char -> int
return cachedHash = hash;
}
}
```

If a String's compute hashCode is 0, even after its hashCode is computed andcached, it will be recomputed (because with the != 0 test, Java cannot tellthe difference between a hash code that has not been computed and a hash codethat has been computed with value 0). Typically, the only String whose hashcode is 0 is ""; most other Strings will have a non-0 hashCode. Recomputing thehash code of "" is very quick, because it stores no values (chars.length is 0,so the loop immediately exits). We could include an extra boolean instancevariable named hashCached, initialized to false and set it to true aftercaching. So we would have

```
public int hashCode()
{
if (hashCached)
return cachedHash;
int hash = 0;
for (int i = 0; i < chars.length; i++)
{
hash = 31*hash + chars[i];   //promotion of char -> int
hashCached = true;
return cachedHash = hash;
}
}
```

IV. CONCLUSION

The load is defined as the objects peers or links. Object is defined as the part of the information stored in a system and its popularity is frequency which it is accessed. The object load can be therefore be defined by its size and popularity. Each peer consists of capacity processing time or bandwidth. The request loads is generated by the queries received for object stored locally. This covers sending an receiving message that is the communication cost and computational power for requesting process. When the messages an forwarded from one peer to another peers they are exposed to a routing loads for queries during the information lookup and they only traverse them. And the traffic loads usually consist of both communication and computational power. The loads balancing in the peer to peer system are mainly based upon DHTs. The DHT are generally made to be well balanced under a flow of request and the request must be uniform

REFERENCE

[1]. Felber, P. ; Kropf, P. ; Schiller, E. ; Serbu, S.,"Survey on Load Balancing in Peer-to-Peer Distributed Hash Tables ",in Communications Surveys & Tutorials, IEEE (Volume:PP , Issue: 99),09 ,July 2013

[2]. Jingsha He ; Fujitsu Labs. of America Inc., Sunnyvale, CA, USA ,"An architecture for wide area network load balancing ",Communications, 2000. ICC 2000. 2000 IEEE International Conference on (Volume:2) ,in 2000.

[3]. Zoels, S. ; Inst. of Commun. Networks, Tech. Univ. Munchen, Munich ; Despotovic, Z. ; Kellerer, W.,"Load balancing in a hierarchical DHT-based P2P system ",in Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on 12-15 Nov. 2007

[4]. Serrano, E.J. ; Dept. of Automotive Sci., Kyushu Univ., Fukuoka, Japan ,"On the design of special hash functions for multiple hash tables ",Electrical Engineering, Computing Science and Automatic Control (CCE), 2012 9th International Conference, 26-28 Sept. 2012.

[5]. Bassalygo, L.A. ; Inst. for Problems of Inf. Transmission, Moscow, Russia ; Burmester, M. ; Dyachkov, A. ; Kabatianski, G.,"Hash codes ",Information Theory. 1997. Proceedings., 1997 IEEE International Symposium,29 Jun-4 Jul 1997 .

[6]. Asaka, T. ; NTT Service Integration Labs., Tokyo, Japan ; Miwa, H. ; Tanaka, Y.."Distributed Web caching using hash-based query caching method ",Control Applications, 1999. Proceedings of the 1999 IEEE International Conference on (Volume:2) ,1999.

[7]. Shie-Yuan Wang ; Dept. of Comput. Sci., Nat. Chiao Tung Univ., Hsinchu, Taiwan ; Chih-Che Lin ; Chao-Chan Huang,"The effects of underlying physical network topologies on peer-to-peer application performances ",Personal Indoor and Mobile Radio Communications (PIMRC), 2010 IEEE 21st International Symposium ,26-30 Sept. 2010 .

[8]. Wei Li ; Beijing Univ. of Posts & Telecommun., Beijing ; Shanzhi Chen ; Tao Yu,"UTAPS: An Underlying Topology-Aware Peer Selection Algorithm in BitTorrent ",Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference ,25-28 March 2008 .

[9]. Dougherty, M. ; Comput. Sci. Dept., East Stroudsburg Univ. of Pennsylvania, East Stroudsburg, PA ; Kimm, H. ; Ho-sang Ham,"Implementation of the Distributed Hash Tables on Peer-to-peer Networks",Sarnoff Symposium, 2008 IEEE ,28-30 April 2008.

