Distributed Hash Table: Blessing or Burden?

October, 2008

Abstract

Peer to Peer (P2P) applications are one of the important kinds of applications now-a-days. Decentralized and distributed nature is the key feature of these P2P applications. Distributed nature implies that one node has to depend on other nodes to complete a task. This task can vary depending on the particular P2P application, but common problem most of the P2P applications face is how to find any data item in the large P2P applications in a scalable manner. The problem which is normally mentioned as “lookup problem” is the heart of any P2P application. Distributed Hash Table (DHT) provides a solution of this problem in an efficient, robust and scalable way. But complexities of implementing truly scalable distributed DHT have raised the question of worthiness of using it for practical purposes.

I have taken the position in support of DHT. I will justify my position first by indicating the drawbacks of existing alternatives of DHT and then will give a brief description DHT and finally will show how DHT handles most of the burning issues regarding distributed look up.

1. Introduction

P2P network which is an overlay network on top of another underlying network is organized under an independent logic. In P2P system there is no centralized control, every node acts both as client and server. That means each node doesn’t only take service from the system but also provides services to other nodes. For example, the most common types of P2P systems are file sharing systems, where instead of a server providing the files, each node provides the files for others. Not only for file sharing applications, for other P2P applications like distributed file system, database, content distribution, naming systems, chat services etc decentralization of data is an integral part. As a result, common question to ask how a node locates a particular piece of data among all the other nodes in the system. One of the answers of this question is Distributed Hash Table. In the next section, I will first define the “Look Up” problem and then in the later sections go through the alternate solutions of this problem and their drawbacks. Finally I will try to give a brief overview of generic DHT and depict how it handles various issues of distributed system.

2. The Lookup Problem

We can state the lookup problem as: Given a data item X stored at some dynamic set of nodes in system, find it. This problem is an imperative one in many distributed systems and is the critical common problem in P2P systems.

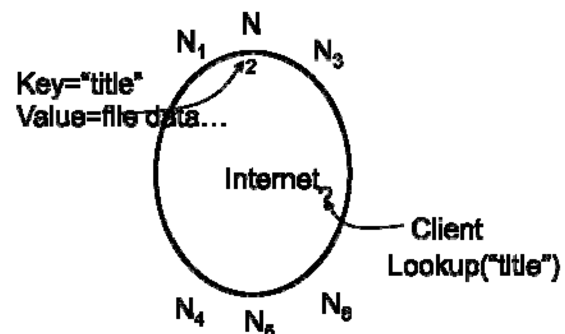

Fig 1: Lookup Problem

As we can see in the Fig 1, node N2 stores the data item indentified by key "title". Any nodes that need that piece of data look it up in the internet using the key "title".

3. Approaches and Their Drawbacks

Before digging into Distributed Hash Table (DHT) let see first what are the alternatives of DHT, how they solve the "look up" problem and what kind of drawbacks they have. Later we will see how DHT overcomes those drawbacks to provide a robust, fault tolerant building block of distributed system.

Centralized Lookup Approach:

As its name implies, in this approach system maintains a central database which maps a file name to the locations of nodes that store the file. For example, Napster [1] adopted this approach for song titles. It had a central index server which maintained the map of song title to the node which held the song. Each node upon joining would send a list of locally held files to the lookup server. When a node would look for a particular song, it would query for it in the lookup server. Lookup server would perform searches and provide the nodes address to the querier. This approach seems very simple, but it violates the main theme of distributed system by using a centralized component. This central component is prone to number of problems. For example, it has lack of tolerance to faults, high congestion and low scalability. It was also responsible for the lawsuits [2] against Napster which sadly caused it to shut down.

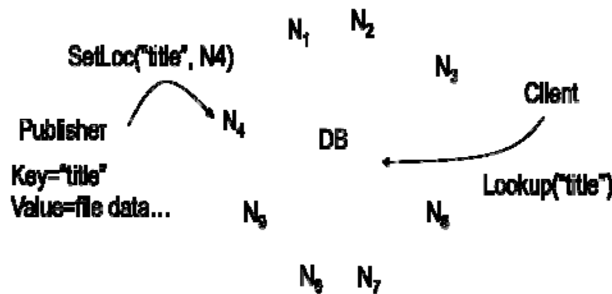

Fig 2: Centralized Lookup Approach

In Fig 2, we can see the centralized approach. Here node N4 stores the value for the key "title". Whenever some clients want to get the data indentified by key "title", they query to the central database DB which sends back the location of N4.

Hierarchical Approach:

The traditional approach for achieving scalability is to use hierarchy. The Internet's Domain Name System (DNS) does this for name lookups. Searches start at the top of the hierarchy and, by following forwarding references from node to node, traverse a single path down to the node containing the desired data. The disadvantage of this approach is that here nodes are not symmetric; failure or removal of the root or a node sufficiently high in the hierarchy can be catastrophic, and the nodes higher in the tree take a larger fraction of the load than the leaf nodes.

Broadcast Flooding Approach:

The previous two approaches are not representative of symmetric lookup as there some nodes are more important than the others. On the other hand, in Broadcast Flooding Approach all nodes are equally important. In this approach, every node just knows its neighbours addresses. The querier node broadcasts a message to all its neighbours with a request for some data X.

When a neighbour node receives such a request, it checks its local database. If it contains X, it responds with the item. Otherwise, it forwards the request to its neighbours, which execute the same protocol. Gnutella [3] follows this approach with some mechanisms to avoid request loops. In Fig 3 we can see the Gnutella approach. Client sends the query to its neighbours. Upon receiving the query, neighbours do the same thing until it reaches to the desired node.

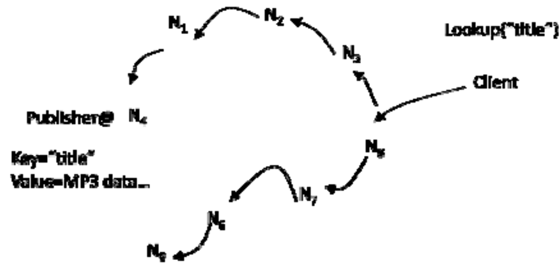

Fig 3: Gnutella Approach

This approach is must more robust than the previous two approaches. But lookup performance suffers because in worst case it requires O(N) messages per lookup where N is the number of nodes in the system. It doesn't scale well because of the bandwidth consumed by broadcast messages and the compute cycles consumed by the many nodes that must handle these messages. In fact, the day after Napster was shut down, reports indicated the Gnutella network collapsed under the load created by a large number of users who migrated to it for sharing music.

One approach for handling such scaling problems is to add "superpeers" in a hierarchical structure, as is done in FastTrack's[4] P2P platform and has been popularized by applications like KaZaa[5]. However, this comes at the expense of resilience to failures of super peers near the top of the hierarchy. Furthermore, this approach does not provide guarantees on object retrieval.

FreeNet[6] Approach:

Freenet uses a key-based symmetric lookup strategy similar to distributed hash table. Here, queries are forwarded from node to node until the desired object is found based on unstructured routing tables dynamically built up using caching. Here, each node not only transfers the data to the queried node but also caches them; as a result forms a huge distributed cache. One of the key features of Freenet is anonymity—creates some challenges for the system. To ensure anonymity, Freenet avoids associating a document with any predictable node, or forming a predictable topology among nodes. As a result, unpopular documents may simply disappear from the system, since no node has the responsibility for maintaining replicas. Furthermore, a search may often need to visit a large fraction of nodes in the system, and no guarantees are possible. Another disadvantage is that as they use encryption, node operators have no control over what kind of content is stored on their node.

We have seen each approaches have some pitfalls. Some of them are not robust while others are not scalable. Now we will discuss DHT approach in details and will show how it overcomes most of the drawbacks faced by other approaches.

4. What is Distributed Hash Table (DHT)?

We can think Distributed Hash Table as a huge dictionary where the dictionary doesn't reside in a single node, rather it is scattered among all the participant nodes. Hash table (dictionary) stores key-value pairs, e.g. ("rutgers-cs", "www.cs.rutgers.edu"), where "rutgers-cs" is the key, while "www.cs.rutgers.edu" is the value. A user of the dictionary must use a key to store or to retrieve the corresponding value. In centralized hash table keys are mapped to array positions by a hash function. But in DHT keys are mapped to particular nodes which contain the corresponding value of the key. DHT hides the underlying complexity and provides a simple lookup interface for DHT applications, as we can see in the fig 4.

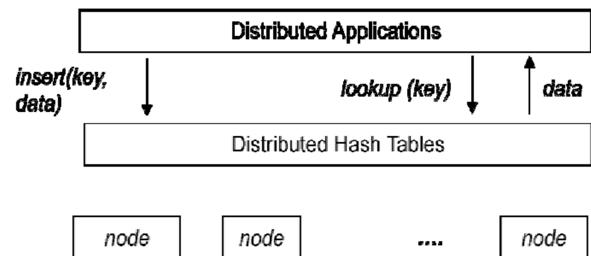

Fig 4: DHT lookup

A DHT implements just one operation: lookup(key) yields the network location of the node currently responsible for the given key. A simple distributed storage application might use this interface as follows. To publish a file under a particular unique name, the publisher would convert the name to a numeric key using an ordinary hash function such as SHA-1, then call lookup(key). The publisher would then send the file to be stored at the node(s) responsible for the key. A consumer wishing to read that file would later obtain its name, convert it to a key, call lookup(key), and ask the resulting node for a copy of the file.

5. Routing Scheme of a Generic DHT

DHT uses a globally known hash function which converts the key to a pseudo-random value, e.g. an integer or position in a virtual space. This hash function must have the properties of balancing out the distribution of keys throughout the key space. DHT creates an underlying graph G, which is a function of the set of existing nodes. Each node in the graph G is identified by a pseudo random identifier, where the space of identifiers of the nodes must coincide with the output space of hash functions. It is important to distribute the nodes and keys evenly in the space, otherwise some nodes will be overloaded and others will be underused.

The routing scheme must create a connected graph where each node can reach other nodes. To retrieve (or to store) the value corresponding to a key, the DHT must find the node that stores that key. For example, assume key “rutgers-cs” is hashed to 109. If there is a node with identifier 109 then that node should be responsible for that key. If such node doesn’t exist, some other node closer to 109 in the space will have the responsibility. Assume node with identifier 112 has this responsibility. Any node, assume node 20, looking for the website address of Rutgers CS department, will first hash the key “rutgers-cs” using the same hash function. So, it will end up getting the hash value 109. In general, nodes only have a partial view of the network and node 20 will not know whether node 109 exists or not. Nevertheless, DHT will try to route the message to node 109 and will always end up in node 111. To reduce the path length DHTs use routing table where they not only store neighbours addresses but also addresses of distant nodes.

Dynamic node management is important for DHTs. Entrance and departure of nodes from DHT network is often very similar from one DHT to the next. We are taking Pastry [7] as example:

Entrance: To enter the network, a node first request DHT to provide it an identifier. For example, the identifier is N. Then new node will ask any node N_{any} in the network to send a JOIN message to node N (which hopefully does not exist in the network). This JOIN message will end up to a node N_{res} which was responsible for the identifier N. Upon receiving the JOIN message node N_{res} realizes that there is a new neighbour coming and it divides its own space and its keys with the newcomer. The remaining actions to take strongly depend on the concrete DHT.

Departure: Ideally, before departure deportee node should inform its neighbours about its departure. Its keys are than redistributed among neighbours (depends on particular DHT). But, in practice nodes leave abruptly, which makes redistribution option impossible. Normally, DHT uses replication to overcome this problem.

6. Challenges in Distributed Environment and DHT Approach

So far, we have seen what Distributed Hash Table (DHT) is and how generally they are handling routing. This section we will see the challenges DHT faces in distributed environment and what different approaches they have taken to overcome the challenges.

a. Scalable Lookup: In section 3, we have seen that most of the alternative approaches for lookup don’t scale well with increasing number of nodes. So it is very important for DHT to scale well. To ensure scalability all DHTs consider the following issues:

Load-balanced mapping of keys to nodes

It is very important to map the keys to nodes in a load balanced way. In general, all keys and nodes are indentified using an m-bit number or identifier (ID). For example, in Chord[8], for their experiments they have used 160 bits keys obtained from SHA-1[9] cryptographic hash function. Each key is stored at one or more nodes whose IDs are "close" (distance function is used to measure the closeness) to the key in the ID space. Most of the DHTs also use caching for popular keys. For example, a node P is responsible for keeping election related information. Prior to election, this node will become a hotspot, which will affect the overall performance and bandwidth of the system. To overcome this scenario, besides forwarding, nodes along the path also store popular keys. This works well as only O(logN) nodes have fingers pointing to P.

Forwarding a lookup for a key to an appropriate node

DHT makes sure that any node that receives a query for a key identifier s forwards it to a node whose ID is "closer" to *s*. This rule guarantees that the query eventually arrives at the closest node. For example, Chord uses finger table for forwarding purpose.

Distance function

The two previous issues allude to the "closeness" of keys to nodes and nodes to each other; this is measured with the help of a distance function whose definition depends on the particular DHT scheme. In Chord, the closeness is the numeric difference between two IDs; in Pastry and Tapestry [10], it is the number of common prefix bits; in Kademlia [11], it is the bit-wise exclusive-or (XOR) of the two IDs. In all the schemes, each forwarding step reduces the closeness between the current node handling the query and the sought key.

Dynamic routing tables

DHTs use routing table for forwarding query to the correct node. There is a trade-off between routing table size and path length size. Normally lookup scales well if both the routing table size and path length size is logarithmic with respect to number of nodes. Most of the DHT ensure this feature. For example, Chord requires O(logN) memory for routing table, at the same time it requires to traverse O(logN) hops for finding the correct node. It is vital to make sure that joining or departure of a node doesn't disrupt all the nodes in the system rather it affects only to a limited number of nodes. In most of the DHTs, this effect is local rather than global. It requires routing table update of limited number of nodes and key distribution among few neighbour nodes.

b. Handling Failures

DHT has to make sure that the overall system will not break down in case of node failures. If a node fails, keys and values the node was holding will be lost. To prevent this to happen most of the DHTs use redundancy. Instead of knowing only their neighbours IP address, each node knows IP addresses of next r nodes and key is replicated in all those r nodes. Now system will not fail unless all r nodes fail.

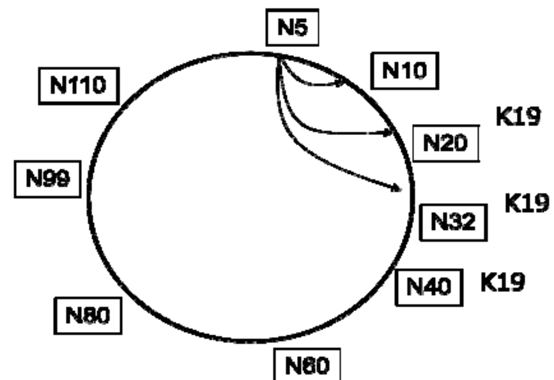

Fig 5: Key Replication

As we can see in Fig 5, K19 is kept not only in N20 but also in N32 and N40. Here, replication degree is 3. So this system will fail only all these

3 nodes fail. Routing table needs to keep more information also. If forwarding to a particular node fails, it can try the other replicas or instead of forwarding the query to one node it can simultaneously forward the query to all r nodes. Here, the trade-off is between latency and bandwidth.

c. Optimize routing to reduce latency

One of the problems most DHTs faced initially was that overlay networks introduced by them bore only a limited relation with the underlying network. As a result each hop in the DHT may have represented a very long distance in the Internet and successive hops may have made the message travel back and forth several times. For example, in Chord nodes of the system form a ring. Proximities of nodes are based on the distance of their identifiers. In the previous versions of Chord algorithm, assignment of identifiers to nodes and distance function was totally independent of the underlying network. As a result a node close in the ring may be thousands mile away in internet which would definitely cause latency. But now most of the DHTS are aware of this issue. CAN[12], Kademlia, Pastry and Tapestry have heuristics to choose routing-table entries referring to nodes that are nearby in the underlying network. Even new version of Chord uses an algorithm proposed by Karger and Ruhl for proximity routing [13].

d. Security

Security is a big concern in any system. Specially, in a distributed system where there is no centralized component, malicious attack is more prevalent. DHTs have already been a victim of various kinds of attacks. As in most of the DHTs there were no restriction over the joining of nodes in the system, bad nodes could easily join and influence the routing. It has introduced security issues like incorrect lookup, denial of service and inconsistent behavior. For example, consider in the network in fig 6, node N99 is a bad node and it has the responsibility for the key range from 81 to 99. One of the attacks it can accomplish is just by denying the existence of the keys. It can also affect the routing behavior. For example, node N10 has asked N99 for key 108. When N99 is supposed to forward the request to N110, it could just forward the request to N80. Now if there are two bad nodes in the system, they could easily create a loop by forwarding queries to each other every time.

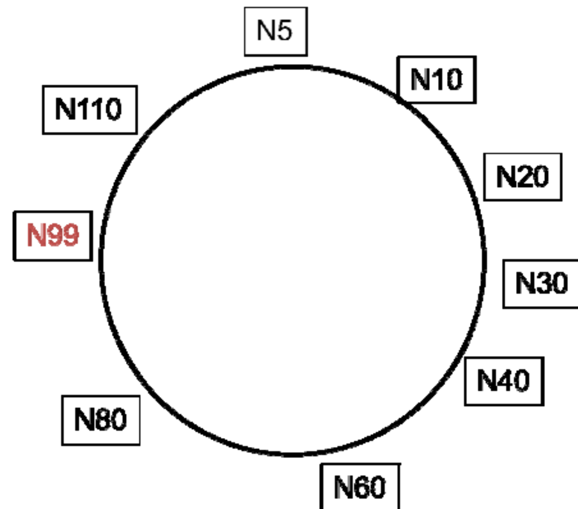

Fig 6: DHT network with bad node

One of the ways to get rid of this attack is to use redundancy. In this way, bad ways can be bypassed. But again problem will occur when there are enough bad nodes to overturn replication. Sybil[14] did exactly that. It creates multiple identities and controls enough nodes to foil redundancy. Various solutions are proposed for overcoming these kinds of attack. One approach is to use secure node ID. Certificate authority provides public key only to good nodes. Every node signs and verifies message. There are also other approaches which exploit techniques like Byzantine protocol [15], weak secure ID etc.

7. DHT Applications

In the previous section we have seen what kinds of challenges DHT have been faced and how it has risen above all of them. In this section we will see where DHT is being used and how they have proved their worthiness.

DHT can be used in any distributed system where there is a need for lookup. As most of the distributed applications require lookup, DHT is now being used in various kinds of applications.

Approach	Abstraction	Easy Use	Scalability	Load Balance	Fault-tolerance	Self-Org	Admin
DHT	High	High	High	Yes	High	Yes	Low
Centralized Lookup	Medium	Medium	Low	No	Low	No	Medium
Broadcast Flooding	Medium	Medium	Low	No	Depends	Yes	Low

Fig 7: Comparison Chart

Below we can see some of the common applications using DHT.

- File sharing
  - Cryptographic File System(CFS) [16]
  - OceanStore[17]
  - Wayfinder[18]
- Web cache
  - Squirrel
- Naming systems
  - ChordDNS
  - INS[
- Censor-resistant stores
  - Eternity
  - FreeNet
- Event notification
  - Scribe
- Query and indexing
  - Kademlia
- Backup store
  - HiveNet
- Web archive
  - Herodotus

8. Comparison

We have seen both DHT and alternative approaches for lookup. Now is the good time for providing a comparison chart to summarize the dominance of DHT over other approaches. Fig 7 provides this comparison chart. As we can see in the chart DHT comes out top in most of the issues.

9. Conclusion

It was not a hard decision for me to take the position in favor of DHT. DHT provides a robust, scalable, self-organized, balanced and fault-tolerant building block using that complex distributed application can be built.

Acknowledgement

I should thank Professor Rich for providing valuable information in the class. Advance thanks to the reviewers for their time and suggestion.

References

[1] Napster: http://en.wikipedia.org/wiki/Napster

[2] Napster faces Shutdown: http://www.wired.com/politics/law/news/2001/02/41752

[3] Gnutella: http://en.wikipedia.org/wiki/Gnutella

[4] FastTrack: http://en.wikipedia.org/wiki/FastTrack

[5] Kazaa: http://en.wikipedia.org/wiki/Kazaa

[6] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, “Freenet: A Distributed Anonymous Information Storage and Retrieval System”

[7] Antony Rowstron1 and Peter Druschel. “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”

[8] Ion Stoic, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, "Chord: A Scalable Peer to peer Lookup Service for Internet Applications"

[9] SHA-1: http://en.wikipedia.org/wiki/SHA

[10] Zhao, B.Y.; Ling Huang; Stribling, J.; Rhea, S.C.; Joseph, A.D.; Kubiatowicz, J.D., "Tapestry: a resilient global-scale overlay for service deployment"

[11] Petar Maymounkov, David Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric"

[12] Sylvia Ratnasamy, Paul Francis2 Mark Handley, Richard Karp, Scott Shenker,"A Scalable Content-Addressable Network"

[13] Karger, K., and Ruhl M., Finding nearest neighbors in growth restricted metrics. In Proceedings ACM Symp. on the Theory of Computing (May 2002), 741–750.

[14] John R. Douceur,Microsoft Research, "The Sybic Attack"

[15] L. Lamport, R. Shostak, and M. Pease (July 1982). "The Byzantine Generals Problem", ACM Trans. Programming Languages and Systems 4 (3): 382–401.

[16] Matt Blaze, AT&T Bell Laboratories, "A Cryptographic File System for Unix".

[17] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski,Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage"

[18] C. Peery, F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen, "Wayfinder: Navigating and Sharing Information in a Decentralized World." in Databases, Information Systems, and Peer-to-Peer Computing
- Second International Workshop, (DBISP2P), 2004