Distributed Hash Tables, Part I

Software

by Brandon Wiley on October 1, 2003

In the world of decentralization, distributed hash tables (DHTs) recently have had a revolutionary effect. The chaotic, ad hoc topologies of the first-generation peer-to-peer architectures have been superseded by a set of topologies with emergent order, provable properties and excellent performance. Knowledge of DHT algorithms is going to be a key ingredient in future developments of distributed applications.

A number of research DHTs have been developed by universities, picked up by the Open Source community and implemented. A few proprietary implementations exist as well, but currently none are available as SDKs; rather, they are embedded in commercially available products. Each DHT scheme generally is pitched as being an entity unto itself, different from all other schemes. In actuality, the various available schemes all fit into a multidimensional matrix. Take one, make a few tweaks and you end up with one of the other ones. Existing research DHTs, such as Chord, Kademlia and Pastry, therefore are starting points for the development of your own custom schemes. Each has properties that can be combined in a multitude of ways. In order to fully express the spectrum of options, let's start with a basic design and then add complexity in order to gain useful properties.

Basically, a DHT performs the functions of a hash table. You can store a key and value pair, and you can look up a value if you have the key. Values are not necessarily persisted on disk, although you certainly could base a DHT on top of a persistent hash table, such as Berkeley DB; and in fact, this has been done. The interesting thing about DHTs is that storage and lookups are distributed among multiple machines. Unlike existing master/slave database replication architectures, all nodes are peers that can join and leave the network freely. Despite the apparent chaos of periodic random changes to the membership of the network, DHTs make provable guarantees about performance.

To begin our exploration of DHT designs, we start with a circular, double-linked list. Each node in the list is a machine on the network. Each node keeps a reference to the next and previous nodes in the list, the addresses of other machines. We must define an ordering so we can determine what the “next” node is for each node in the list. The method used by the Chord DHT to determine the next node is as follows: assign a unique random ID of k bits to each node. Arrange the nodes in a ring so the IDs are in increasing order clockwise around the ring. For each node, the next node is the one that is the smallest distance clockwise away. For most nodes, this is the node whose ID is closest to but still greater than the current node's ID. The one exception is the node with the greatest ID, whose successor is the node with the smallest ID. This distance metric is defined more concretely in the distance method (Listing 1).

Listing 1. ringDistance.py

```
# This is a clockwise ring distance function.
# It depends on a globally defined k, the key size.
# The largest possible node id is 2**k.
def distance(a, b):
    if a==b:
        return 0
    elif a<b:
        return b-a;
    else:
        return (2**k)+(b-a);
```

Each node is itself a standard hash table. All you need to do to store or retrieve a value from the hash table is find the appropriate node in the network, then do a normal hash table store or lookup there. A simple way to determine which node is appropriate for a particular key (the one Chord uses) is the same as the method for determining the successor of a particular node ID. First, take the key and hash it to generate a key of exactly k bits. Treat this number as a node ID, and determine which node is its successor by starting at any point in the ring and working clockwise until a node is found whose ID is closest to but still greater than the key. The node you find is the node responsible for storage and lookup for that particular key (Listing 2). Using a hash to generate the key is beneficial because hashes generally are distributed evenly, and different keys are distributed evenly across all of the nodes in the network.

Listing 2. findNode.py

```
# From the start node, find the node responsible
# for the target key
def findNode(start, key):
    current=start
    while distance(current.id, key) > \
          distance(current.next.id, key):
        current=current.next
    return current

# Find the responsible node and get the value for
# the key
def lookup(start, key):
    node=findNode(start, key)
    return node.data[key]

# Find the responsible node and store the value
# with the key
def store(start, key, value):
    node=findNode(start, key)
    node.data[key]=value
```

This DHT design is simple but entirely sufficient to serve the purpose of a distributed hash table. Given a static network of nodes with perfect uptime, you can start with any node and key and find the node responsible for that key. An important thing to keep in mind is that although the example code so far looks like a fairly normal, doubly linked list, this is only a simulation of a DHT. In a real DHT, each node would be on a different machine, and all calls to them would need to be communicated over some kind of socket protocol.

In order to make the current design more useful, it would be nice to account for nodes joining and leaving the network, either intentionally or in the case of failure. To enable this feature, we must establish a join/leave protocol for our network. The first step in the Chord join protocol is to look up the successor of the new node's ID using the normal lookup protocol. The new node should be inserted between the found successor node and that node's predecessor. The new node is responsible for some portion of the keys for which the predecessor node was responsible. In order to ensure that all lookups work without fail, the appropriate portion of keys should be copied to the new node before the predecessor node changes its next node pointer to point to the new node.

Leaves are very simple; the leaving node copies all of its stored information to its predecessor. The predecessor then changes its next node pointer to point to the leaving node's successor. The join and leave code is similar to the code for inserting and removing elements from a normal linked list, with the added requirement of migrating data between the joining/leaving nodes and their neighbors. In a normal linked list, you remove a node to delete the data it's holding. In a DHT, the insertion and removal of nodes is independent of the insertion and removal of data. It might be useful to think of DHT nodes as similar to the periodically readjusting buckets used in persistent hash table implementations, such as Berkeley DB.

Allowing the network to have dynamic members while ensuring that storage and lookups still function properly certainly is an improvement to our design. However, the performance is terrible—O(n) with an expected performance of n/2. Each node traversed requires communication with a different machine on the network, which might require the establishment of a TCP/IP connection, depending on the chosen transport. Therefore, n/2 traversed nodes can be quite slow.

In order to achieve better performance, the Chord design adds a layer to access O(log n) performance. Instead of storing a pointer to the next node, each node stores a “finger table” containing the addresses of k nodes. The distance between the current node's ID and the IDs of the nodes in the finger table increases exponentially. Each traversed node on the path to a particular key is closer logarithmically than the last, with O(log n) nodes being traversed overall.

In order for logarithmic lookups to work, the finger table needs to be kept up to date. An out-of-date finger table doesn't break lookups as long as each node has an up-to-date next pointer, but lookups are logarithmic only if the finger table is correct. Updating the finger table requires that a node address is found for each of the k slots in the table. For any slot x, where x is 1 to k, finger[x] is determined by taking the current node's ID and looking up the node responsible for the key $(id+2^{(x-1)}) \bmod (2^k)$ (Listing 3). When doing lookups, you now have k nodes to choose from at each hop, instead of only one at each. For each node you visit from the starting node, you follow the entry in the finger table that has the shortest distance to the key (Listing 4).

Listing 3. update.py

```
def update(node):
    for x in range(k):
        oldEntry=node.finger[x]
        node.finger[x]=findNode(oldEntry,
                                (node.id+(2**x)) % (2**k))
```

Listing 4. finger-lookup.py

```
def findFinger(node, key):
    current=node
    for x in range(k):
        if distance(current.id, key) > \
           distance(node.finger[x].id, key):
            current=node.finger[x]
    return current

def lookup(start, key):
    current=findFinger(start, key)
    next=findFinger(current, key)
    while distance(current.id, key) > \
          distance(next.id, key):
        current=next
        next=findFinger(current, key)
    return current
```

So far we have more or less defined the original version of the Chord DHT design as it was described by the MIT team that invented it. This is only the tip of the iceberg in the world of DHTs, though. Many modifications can be made to establish different properties from the ones described in the original Chord paper, without losing the logarithmic performance and guaranteed lookups that Chord provides.

One property that might be useful for a DHT is the ability to update the finger table passively, requiring periodic lookups to be done in order to refresh the table. With MIT Chord, you must do a lookup, hitting O(log n) nodes for all k items in the finger table, which can result in a considerable amount of traffic. It would be advantageous if a node could add other nodes to its finger table when they contacted it for lookups. As a conversation already has been established in order to do the lookup, there is little added overhead in checking to see if the node doing the lookup is a good candidate for the local finger table. Unfortunately, finger table links in Chord are unidirectional because the distance metric is not symmetrical. A node generally is not going to be in the finger tables of the nodes in its finger table.

A solution to this problem is to replace Chord's modular addition distance metric with one based on XOR. The distance between two nodes, A and B, is defined as the XOR of the node IDs interpreted as the binary representation of an unsigned integer (Listing 5). XOR makes a delightful distance metric because it is symmetric. Because distance(A, B) ==

distance(B, A), for any two nodes, if A is in B's finger table then B is in A's finger table. This means nodes can update their finger tables by recording the addresses of nodes that query them, reducing significantly the amount of node update traffic. It also simplifies coding a DHT application, because you don't need to keep a separate thread to call the update method periodically. Instead, you simply update whenever the lookup method is called.

Listing 5. xor-distance.py

```
def distance(a, b):
    return a^b # In Python, this means a XOR b,
               # not a to the power of b.
```

An issue with the design presented so far is the paths to a given node are fragile. If any node in the path refuses to cooperate, the lookup is stuck. Between any two nodes there is exactly one path, so routing around broken nodes is impossible. The Kademlia DHT solves this by widening the finger table to contain a bucket of j references for each finger table slot instead of only one, where j is defined globally for the whole network. Now j different choices are available for each hop, so there are somewhere around j*log(n) possible paths. There are less than that, though, paths converge as they get closer to the target. But, the number of possible paths probably is greater than 1, so this is an improvement.

Kademlia goes further and orders the nodes in the bucket in terms of recorded uptime. Older nodes are given preference for queries, and new references are added only if there are not enough old nodes. Besides the increased reliability of queries, this approach offers the added benefit that an attack on the network in which new nodes are created rapidly in order to push out good nodes will fail—it won't even be noticeable.

It's important to understand that these different properties are not tied to a particular DHT implementation. We gradually have built up a DHT design from scratch, developed it into something that resembles Chord, then modified it to be more like Kademlia. The different approaches can be more or less mixed and matched. Your finger table buckets can have 1 slot or j slots, depending on whether you use modular addition or XOR for your distance metric. You can always follow the closest node, or you can rank them according to uptime or according to some other criteria. You can draw from several other DHT designs, such as Pastry, OceanStore and Coral. You also can use your own ideas to devise the perfect design for your needs. Myself, I have concocted several modifications to a base Chord design to add properties such as anonymity, Byzantine fault-tolerant lookups, geographic routing and the efficient broadcasting of messages to enter the network. It's fun to do and easier than you think.

Now that you know how to create your own DHT implementations, you're probably wondering what kind of crazy things you can do with this code. Although there probably are many applications for DHTs that I haven't thought of yet, I know people already are working on such projects as file sharing, creating a shared hard drive for backing up data, replacing DNS with a peer-to-peer name resolution system, scalable chat and serverless gaming.

For this article, I've tied the code together into a fun little example application that might be of interest to readers who caught my interview on the *Linux Journal* Web site about peer-to-peer Superworms (see Resources). The application is a distributed port scanner that stores results in the simulated DHT (Listing 6). Given a fully functional DHT implementation, this script would have some handy properties. First, it allows multiple machines to contribute results to a massive scanning of the Internet. This way, all of the scanning activity can't be linked with a single machine. Additionally, it avoids redundant scanning. If the host already has been scanned, the results are fetched from the DHT, avoiding multiple scans. No central server is required to hold all of the results or to coordinate the activities of the participants. This application may seem somewhat insidious, but the point is it was trivial to write given the DHT library. The same approach can be used in other sorts of distributed projects.

Listing 6. portscan.py

```
def __main__():
    id=int(random.uniform(0,2**k))
    node=Node(id)
    join(node, initialContact)

    line=raw_input('Enter an IP to scan: ').trim()
    key=long(sha.new(line).hexdigest(),16)
    value=lookup(node, key)
    if value==None:
        f=os.popen('nmap '+args[1])
        lines=f.readlines()
        value=string.join(lines, '\n')
        store(node, key, value)
```

In this installment of our two-part series, we discussed the theory behind building DHTs. Next time, we'll talk about practical issues in using DHTs in real-world applications.

Resources

Achord: thalassocracy.org/achord/achord-iptps.html

Chord: www.pdos.lcs.mit.edu/chord

Curious Yellow: blanu.net/curious_yellow.html

How Can You Defend against a Superworm? linuxjournal.com/article/6069

Kademlia: kademlia.scs.cs.nyu.edu

Brandon Wiley is a peer-to-peer hacker and current president of the Foundation for Decentralization Research, a nonprofit corporation empowering people through technology. He can be contacted at blanu@decentralize.org.