CSE202 Project - Consistent Hashing

Neil Alldrin

November 27, 2002

1 Introduction

Consistent hashing is a special kind of hashing which is useful for certain applications. Originally devised by Karger et. al. at MIT[2] for use in distributed caching, the idea has now been expanded to other areas, most notably peer to peer networking[5][4]. As internet use increases and as distributed systems grow more prevalent (ie, Napster clones), consistent hashing could play an increasingly important role.

At a basic level, consistent hashing is just a special kind of hashing. Namely, a consistent hash function is one which changes minimally as the range of the function changes[2]. This makes consistent hashing ideal in circumstances where the set of buckets changes over time. Furthermore, it can be shown that, with high probability, two users with inconsistent but overlapping sets of buckets will map items to the same buckets. This is particularly important because it eliminates the need to maintain consistent state among all nodes in a network.

In this paper I present a family of consistent hash functions, attempt to prove certain properties of these functions, and describe an application of consistent hashing, namely peer to peer networking.[1]

2 Consistent Hashing

2.1 Overview

This section formally defines what a consistent hash family is and describes a possible implementation of a consistent hash function.

2.2 Definitions

A *view* V is the set of buckets of which a particular user is aware. It is assumed that while views can be inconsistent, each user is aware of a constant fraction of the currently available buckets.

A *ranged hash function* f is a function mapping an assignment of items to buckets for every possible view. We define $f(V, i) = f_V(i)$ to be the bucket where item i is assigned under view V. A further restriction is that $f_V(i) \epsilon V$ since items can only be assigned to buckets a user knows about.

A *ranged hash family* F is a family of ranged hash functions.

[1] These "findings" are primarily a summary of [2], [3] and [4]

A *random ranged hash function* is a function drawn at random from a particular ranged hash family.

A ranged hash family is *balanced* if the fraction of items mapped to each bucket is expected to be $O(1/|V|)$.

A ranged hash function is *monotone* if, for all views $V_1 \subseteq V_2$, if $f_{V_2}(i) \epsilon V_1$ then $f_{V_1}(i) = f_{V_2}(i)$.

The *spread* of a system is defined as follows. Let $\{V_1 ... V_V\}$ be a set of views with C distinct buckets among these views where each view has at least C/t buckets. The spread of a ranged hash function f over item i is the number of unique buckets i is mapped to over all the views. The spread of a ranged hash function in general, denoted $\sigma(f)$, is the maximum of the spreads of each item.

The *load*, $\lambda(b)$, of a ranged hash function f and bucket b is the number of items assigned to b over all user's views. The load of a ranged hash function in general is the maximum load over every bucket.

The last four definitions are quantifiable notions of *consistency* for a hash function/family. A consistent hash function is one that is balanced and monotone, while attempting to minimize spread and load.

The unit interval is the interval $[0, 1]$. Throughout this paper we consider the unit interval to be circular, in other words 1 and 0 are considered adjacent points.

2.3 Assumptions

1. C is the total number of buckets in the system.
2. Each view in the system contains at least C/t buckets for some constant t.

2.4 A Consistent Hash Family

Consider the interval $[0, 1]$. Now randomly map each bucket $K \cdot log(C)$ times to the interval.[2] Also randomly map all items to the same interval. We now define the ranged hash function $f_V(i)$ to return the bucket $b \epsilon V$ closest to i in the interval. By choosing different mappings of buckets and items we obtain a family of ranged hash functions.

2.5 Properties of Our Consistent Hash Family

Theorem 2.1. *The ranged hash family described above is monotone.*

Theorem 2.2. *The ranged hash family described above is balanced. For a fixed view V, the probability that $f_V(i) = b$ where $b \epsilon V$ is $O(1/V)$. The probability that $f_V(i) = b$ where b is in any view is $O(t \cdot log(C)/V)$.*

Theorem 2.3. *The ranged hash family described above has spread $\sigma(i) = O(t \cdot log(C))$ with probability greater than $1 - 1/C^{\Omega(1)}$ when the number of items equals the number of buckets and the number of views is proportional to the number of buckets.*

[2] C is the maximum number of buckets in the range and K is some constant

Theorem 2.4. *The ranged hash family described above has load* $\lambda(b) = O(t \cdot log(C))$ *with probability greater than* $1 - 1/C^{\Omega(1)}$ *when the number of items and buckets are as in Theorem 2.3.*

The proofs for these theorems are sketched in [2].

2.6 Implementation

An implementation of a consistent hash function will involve a set of buckets corresponding to a user's view V and a set of items I.[3] Since we don't know the total number of buckets in the system, we assume an upper bound on C. The operations that can be done by the user are $hash(i)$, $addBucket(b)$, and $deleteBucket(b)$.

Prior to using our algorithm, we have to figure out where each bucket $b \epsilon V$ lies in our unit interval and store that information. The location of each bucket is trivial since it simply involves $|K \cdot V \cdot log(C)| = O(V \cdot log(C))$ calls[4] to a random function with each call taking $O(1)$ time.

Storing the bucket locations in a way that allows efficient hashing of items is more complicated. First, note that we want to store the interval over which an item will be mapped rather than the location of the bucket itself. A fairly good way to do this is to use a single balanced binary search tree to store the correspondence between segments of the unit interval and buckets. In this scheme there will be $K \cdot V \cdot log(C)$ intervals, so the search tree will have depth $O(log(V \cdot log\ C)) = O(log(C \cdot log\ C))$. This puts a lower bound of $\Omega(log(C \cdot log\ C))$ on the $hash$ operation.

A better way to store segment/bucket mappings is to divide the unit interval into $K \cdot C \cdot log(C)$ equal-length segments and to have separate search trees for each segment. Since the expected number of buckets in each segment is $O(1)$, the expected time to determine which bucket an item maps to within a segment is also $O(1)$ (the expected depth of each search tree is $log(O(1)) = O(1)$). Finding out which segment an item belongs to also takes $O(1)$ time (segments are of equal size so the segment can be located by dividing the item's location by the number of segments).

Another detail to consider in an implementation of a consistent hash function is the fact that continuous intervals are used but cannot be realized on computers. To get around this we note that we only need to use enough bits to distinguish each point in the interval, or $O(log\ C)$ bits.

Theorem 2.5. $hash(i)$ *can run in* $O(1)$ *expected time.*

Proof. The time to map an item to a bucket is the sum of the time it takes to locate the item's position in the unit interval and the time it takes to figure out which bucket that location corresponds to. Finding an item's position takes $O(1)$ time since it is a single call to a random function that can run in $O(1)$ time. Mapping a location to a bucket takes $O(1)$ expected time if we use the data structure described and analyzed above. Therefore, the overall run-time is $O(1)$. □

[3] It should be understood that multiple users may be operating on some larger set of buckets, but each user uses the same basic algorithm to hash items to buckets, so we consider only a single user's point of view.

[4] Recall that we need to map each bucket $K \cdot log(C)$ times and that $V \geq C/t$.

Theorem 2.6. *addBucket(b) can run in O(log C) expected time.*

Proof. Suppose we are using the multiple search tree data structure described above for mapping segments to buckets and have already hashed all I possible items into their corresponding buckets. At this point we add another bucket to our view. Doing this will add $K \cdot log(C)$ new bucket intervals each changing their surrounding two bucket intervals. This results in $3K \cdot log(C)$ new or modified intervals. If we re-hash every item contained in segments spanned by these intervals we will have a correct hashing of items since items outside these segments are already in their correct buckets. The expected number of segments spanned by a single interval is $O(\frac{K \cdot C \cdot log(C)}{K \cdot V \cdot log(C)}) = O(1)$. The expected number of items in each segment is also $O(1)$. This means we must re-hash the expected $O(log(C))$ items contained in $O(log(C))$ segments. Each hash takes $O(1)$ expected time so the total time taken is $O(log\ C)$. Since we have considered the worst possible case we are done. □

Theorem 2.7. *deleteBucket(b) can run in O(log C) expected time.*

Proof. This proof is almost identical to the proof for Theorem 2.6. The only difference is that the number of intervals to consider is $2K \cdot log(C)$ instead of $3K \cdot log(C)$ since removing an interval introduces no new intervals and modifies the surrounding two intervals. □

3 An Application : Chord

Chord is a peer-to-peer (P2P) networking protocol developed at MIT[5][4]. It serves as the backbone for other research projects, such as the Cooperative File System[1], as well as being a research project in its own right. This section describes Chord, primarily in the context of consistent hashing.

3.1 Peer to Peer Networks

A P2P network at its most basic level is a decentralized network where all nodes have similar functionality. These networks have significant advantages over traditional networks:

1. No central (and expensive) servers are required.
2. Resources on all nodes can be utilized.
3. P2P networks are more robust against certain types of faults.

P2P networks are also a special case of distributed systems. The primary twist is that nodes are continually joining and leaving the network whereas traditional distributed systems have a fixed set of nodes. Because the set of nodes in the network changes, data must be continually moved from node to node if that data is to be persistent. Content addressable networking partially solves this problem by providing a mapping of keys to machines, but its reliance on a key directory makes it difficult to maintain and limits scalability.

3.2 The Chord Protocol

The Chord protocol takes the idea of content addressable networking and makes it work efficiently in a peer-to-peer (P2P) network. This is accomplished by using a variant on consistent hashing, essentially making the entire network one large distributed hash table.

In the Chord framework, every node is a bucket and every key is an item. Nodes and keys are mapped into a space of size 2^m (aka the unit circle) by standard hash functions (aka random functions). Keys are assigned to the nearest node in the clockwise direction around the 2^m "circle". So far this is just the consistent hashing algorithm described in 2.6 with some superficial changes.

The main difference comes in how and where the location of other nodes are stored. This was previously taken care of in a single local data-structure, but this does not suffice for P2P networks since it would either require a central server which violates the principles of P2P networks or it would require each node to have its own copy which would require too much overhead. Chord gets around these problems by storing this information in a distributed fashion. Basically, every node has pointers to the next $K \cdot log(N)$ nodes in the 2^m circle (these are called successors). This allows lookups to occur in $O(N/log(N))$ time by traversing the circle in order. To speed up lookups, each node also has a finger table which stores the m nodes closest to positions $n+2^i$ for $i = 1$ *to* m. This allows much faster lookups ($O(log(N))$) if the finger table is correct.

When a node joins the system it is mapped somewhere in the circle and becomes the new successor to the node immediately behind it while taking on the next node as its own successor. Keys are also given to the new node by its predecessor. When a node leaves the system he bequeaths his keys to his successor and tells his predecessor he's leaving. If a node stops responding for some reason, it will eventually cause predecessors to remove that node from their successor lists.

3.3 Correctness

The Chord system is difficult to disconnect because each node has pointers to $O(log(N))$ other nodes. This means at least $O(log(N))$ nodes have to fail before it is even possible for the network to become disconnected. Even with failures of up to $N/2$ nodes, the system still remains connected with high probability (assuming a random distribution of failures, an adversary could always choose just the right $O(log(N))$ nodes to disconnect a node). [4] goes into a more detailed theoretical analysis of Chord's robustness, taking into account node joins, departures, and failures albeit in a somewhat limited model. What really proves Chord are the test results, which coincide nicely with the theoretical findings. However, even this is not enough because the tests used nowhere near the number of nodes possible on the internet, so it remains to be seen how well Chord performs as a massive scale system.

4 Conclusion

Consistent hashing has been shown to be a useful idea, both in itself and in the context of peer to peer networking. Most of the benefits of consistent hashes translate directly to peer to peer networking (ie, balanced loads, low bucket

exchange, etc) making these systems "better". Furthermore, tests have shown that, at least in limited contexts, these systems work well in practice.

References

[1] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 202–215.

[2] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), ACM Press, pp. 654–663.

[3] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web caching with consistent hashing. In *Proceeding of the eighth international conference on World Wide Web* (1999), Elsevier North-Holland, Inc., pp. 1203–1213.

[4] Liben-Nowell, D., Balakrishnan, H., and Karger, D. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), ACM Press, pp. 233–242.

[5] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and 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* (2001), ACM Press, pp. 149–160.