Simple Load Balancing for Distributed Hash Tables

John Byers* byers@cs.bu.edu
Jeffrey Considine* jconsidi@cs.bu.edu
Michael Mitzenmacher† michaelm@eecs.harvard.edu

Dept. of Computer Science
Boston University
Boston, Massachusetts

EECS
Harvard University
Cambridge, Massachusetts

Abstract

Distributed hash tables have recently become a useful building block for a variety of distributed applications. However, current schemes based upon consistent hashing require both considerable implementation complexity and substantial storage overhead to achieve desired load balancing goals. We argue in this paper that these goals can be achieved more simply and more cost-effectively. First, we suggest the direct application of the "power of two choices" paradigm, whereby an item is stored at the less loaded of two (or more) random alternatives. We then consider how associating a small constant number of hash values with a key can naturally be extended to support other load balancing strategies, including load-stealing or load-shedding, as well as providing natural fault-tolerance mechanisms.

1 Introduction

Distributed hash tables have been proposed as a fundamental building block for peer-to-peer systems [6, 9, 8, 10, 12]. In the current design of distributed hash tables (DHTs), it is conventionally assumed that keys are mapped to a single peer — that peer is then responsible for storing a value associated with the key, such as the contents of a file with a given name. A widely used design to support such a DHT [10] consists of two components: consistent hashing over a one-dimensional space [6] and an indexing topology to quickly navigate this space.

In a basic consistent hashing approach, both peers and keys are hashed onto a one dimensional ring. Keys are then assigned to the nearest peer in the clockwise direction. Servers are connected to their neighbors in the ring (i.e. the ring structure is embedded in the overlay) and searching for a key reduces to traversing the ring. Fast searches are enabled through additional overlay edges spanning larger arcs around the ring; for example, in Chord [10], a carefully constructed "finger table" of logarithmic size enables searches in a logarithmic number of steps.

However, with the naive implementation of consistent hashing described so far, considerable load imbalance can result. In particular, a peer that happens to be responsible for a larger arc of the ring will tend to be assigned a greater number of items.[1] If there are n peers, the maximum arc length for a peer will be $\Theta(\log n/n)$ with high probability, even though the average arc length is $1/n$.

A solution proposed in [10] is for each peer to simulate a logarithmic number of "virtual peers", thus assigning each peer several smaller segments whose total size is more tightly bounded around the expectation $1/n$. While theoretically elegant, virtual peers do not completely solve the load balancing issue. First, even with perfectly uniform assignments of segments to peers, the load need not be well balanced. In the extreme case where there are n items and n peers, this is the standard balls and bins problem, and with high probability one peer will be responsible for $\Theta(\log n/ \log\log n)$ items. Second, the

*Supported in part by NSF grants ANI-0093296 and ANI-9986397.

†Supported in part by NSF grants CCR-9983832, CCR-0118701, CCR-0121154, and an Alfred P. Sloan Research Fellowship.

[1] For now, we will make the unrealistic assumption that all items are of equal size and popularity. Very popular items, or "hot spots", can be specially handled by appropriate replication, as in [6, 10]. Here, we are concerned with balancing load associated with the bulk of less popular items.

proposal to use $O(\log n)$ virtual peers with $O(\log n)$ edges in each finger table leads to $O(\log^2 n)$ edges per peer. Using the numbers of [10], a network of 100,000 peers will need to maintain 400 edges per peer. Although this number is small in terms of memory consumption, maintaining 400 edges per peer incurs a rather hefty messaging cost since each edge will need to be probed at regular intervals to detect failures and to perform general maintenance.

As a practical alternative to virtual peers, we propose the application of the "power of two choices" paradigm [1, 7] to balance load. These methods are used in standard hashing scenarios using bins (chaining) to reduce the maximum bin load with high probability. Using these methods, two or more hash functions are used to pick candidate bins for each item to be inserted. Prior to insertion, the load of each bin is compared and the item is inserted into the bin with the lowest load. Similarly, to search for an item, one applies the hash functions and examines each bin to locate the item. If there are n items, n bins, and $d \geq 2$ hash functions, the maximum load of any bin is only $\log\log n / \log d + O(1)$ with high probability. Moreover, the maximum load is more tightly concentrated around the mean for any number of balls.

Returning to DHTs, suppose that we have each peer represented by just one point in the circle. Each item chooses $d \geq 2$ possible points in the circle, and is associated with the corresponding peer with the least load from these choices. Previous results for the power of two choices cannot be immediately applied, since in this case the probability of a ball landing in a bin is not uniform. Our first contribution is to examine this interesting case, both theoretically and through simulation.

Our second contribution is to apply these methods in the context of the Chord architecture. We present low-overhead searching methods which are compatible with the two choice storage model and then provide a comparative performance evaluation against the virtual peers approach.

Our final contribution is a consideration of the broader impact of having a key map to a small constant number of peers rather than to a single peer. We argue that the power of two choices paradigm facilitates other load balancing methods, such as load-stealing and load-shedding in highly dynamic DHTs, and enables new methods for addressing fault-tolerance.

2 Two Choices

We first consider the following problem, which is interesting theoretically in its own right. Suppose that we have each of n peers represented by just one point in the circle, to avoid the need for multiple finger tables. Then n items are placed sequentially. Each item uses $d \geq 2$ hash functions to choose locations on the circle; each point is associated with the closest peer (in the clockwise direction). The item is then associated with the peer from this set of at most d peers storing the fewest other items; ties are broken arbitrarily. A natural question to help assess the utility of two choices in this setting is whether in this case, we maintain a $\log\log n / \log d + O(1)$ maximum load with high probability.

Theorem 1 *In the setting above, the maximum load is at most* $\log\log n / \log d + O(1)$ *with high probability.*

Our proof (not included for reasons of space) uses the layered induction technique from the seminal work of [1] (see also [7]). Because of the variance in the arc length associated with each peer, we must modify the proof to take this into account. The standard layered induction uses the fact that if there are β_k bins that have load at least k, then the probability each ball causes a bin to reach load $k+1$ is most $(\beta_k/n)^2$. This is used to derive bounds on β_k that hold with high probability for each k. The key insight that is required for our modification is that the largest β_k arcs have total length "not too much longer" than β_k/n. This result therefore applies to other settings in which the bin sizes are non-uniform, or equivalently, when the probabilities associated with selecting bins are non-uniform.

Our results hold regardless of how ties are broken when more than one choice has the same smallest load. Vöcking uses an improved tie-breaking scheme to improve the d-choice balls and bins result [11]; his extension can also be applied here. However, in this setting we have a natural criterion that can be used to break ties: the length of the arcs. Intuitively, choosing the least loaded arc with the smallest length appears best, since that arc is the least likely to obtain further load in the future. Simulations bear out that this tie-breaking approach is better than breaking ties at random, and in fact in simulations, the resulting load balance is better than that achieved using Vöcking's scheme; however, we cannot yet demonstrate this analytically. We employ this "shorter arc" tie-breaking scheme in our subsequent experiments.

Although this theoretical result is for the simplest setting (items have equal weight, and are inserted sequentially), the paradigm of using two choices is generally successful in more complex situations, including weighted items and cases where items enter and leave the system dynamically [7]. We therefore expect good behavior in the more complex peer-to-peer settings; we plan to continue to derive related theoretical results.

3 DHT Implementation

Now we describe the application of this idea to DHTs. Let h_0 be a universally agreed hash function that maps peers onto the ring. Similarly, let $h_1, h_2, \ldots h_d$ be a series of universally agreed hash functions mapping items onto the ring. To insert an item x using d hash functions, a peer first calculates $h_1(x), h_2(x), \ldots, h_d(x)$. Then, d lookups are executed in parallel to find the peers $p_1, p_2, \ldots, p_d$ responsible for these hash values, according to the mapping given by h_0. After querying the load of each peer, the peer p_i with lowest load is chosen to store x. A straightforward, but naive, implementation of a search requires the peer performing the search to again calculate $h_1(x), h_2(x), \ldots, h_d(x)$. The peer then initiates lookups to find the peers associated with each of these d values, of which at least one will successfully locate the key-value pair. While these searches are inherently parallelizable, and thus enable searching in little more time than their classic counterparts, this approach uses a factor of d more network traffic to perform each search, which may be unacceptable.

To reduce the overhead searching for additional peers, we introduce *redirection pointers.* Insertion proceeds exactly as before. But in addition to storing the item at the least loaded peer p_i, all other peers p_j where $j \neq i$ store a redirection pointer $x \rightarrow p_i$. To search for x, a peer now performs a single query, by choosing a hash function h_j at random in an effort to locate p_i. If p_j does not have x, then p_j forwards the query using a redirection pointer $x \rightarrow p_i$. Lookups now take at most one more step; if h_j is chosen uniformly at random from the d choices, the extra step is necessary with probability $(d-1)/d$. Although this incurs the overhead of keeping these additional pointers, unless the items stored are very small or inexpensive to calculate, storing actual items and any associated computation will tend to dominate any stored pointers.

One hazard with this approach is that the use of explicit redirection pointers introduces a dependence on a particular peer staying up. We assume that a soft state approach [4] is used and the provider of the key periodically re-inserts it, both to ensure freshness and to recover from failures. Replication to nearby peers as in DHash [5] will allow recovery, but a new search will need to be performed to find the replicating peers. This is easily remedied by keeping pointers to some or all of the replicating peers, and similarly, replicating those pointers.

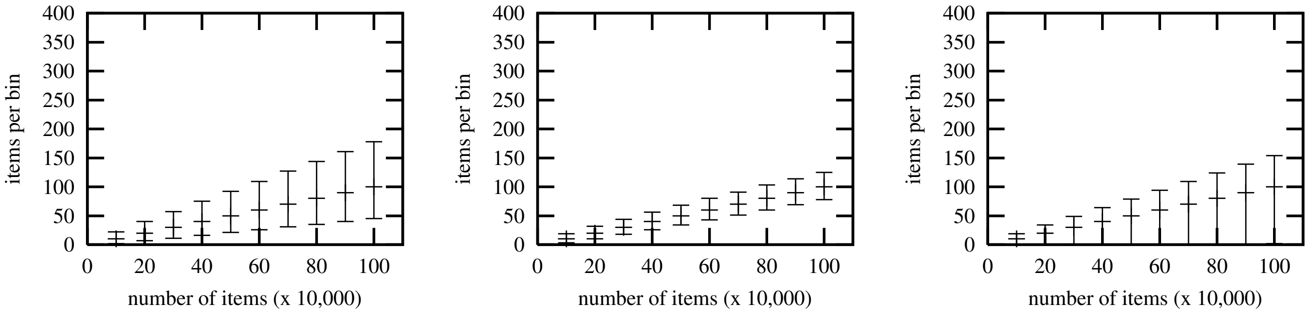
4 Other Virtues of Redirection

While using two or more choices for placement improves load balancing, it still forces a static placement of the items, which may lead to poor performance when the popularity of items changes over time. As mentioned earlier, one means of coping with this issue is to use soft state and allow items to change location when they are re-inserted if their previous choice has become more heavily loaded.[2] However, since redirection pointers give the peers responsible for a key explicit knowledge of each other, they can be used to facilitate a wide range of load balancing methods that react more quickly than periodic re-insertion allows. We briefly explore some of these possibilities here.

Load-stealing and load-shedding become simple in this context. For example, consider load-stealing, whereby an underutilized peer p_1 seeks out load to take from more heavily utilized peers. In the case where items are placed using multiple choices, a natural idea is to have p_1 attempt to steal items for which p_1 currently has a redirection pointer. This maintains the invariant that an item is associated with one of its d hash locations. Alternatively, the stealing peer could break this invariant, but at the risk of additional implementation complexity. In general, a load-stealing peer could identify an arbitrary peer p_2 and take responsibility for an item x by making a replica of x and having p_2 create a redirection pointer to p_1 for item x.

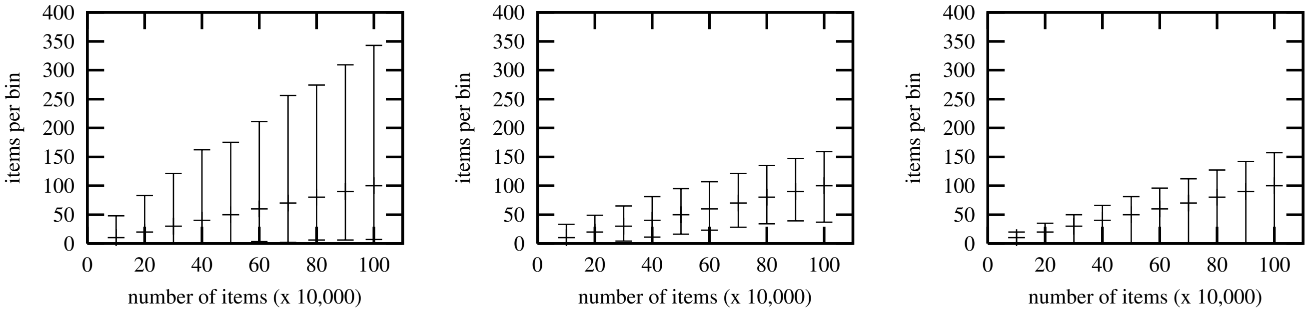
Load-shedding, whereby an overloaded peer attempts to offload work to a less loaded peer, is also

[2] This is essentially a dynamic balls and bins problem [7].



(a) virtual peers (b) unlimited virtual peers (c) two choices

Figure 1: 1st percentile, mean and 99th percentile loads using various load balancing strategies.



(a) virtual peers (b) unlimited virtual peers (c) two choices

Figure 2: Minimum, mean and maximum loads using various load balancing strategies.

well suited to peer-to-peer networks. An overloaded peer p_1 must pass on an item x and create a redirection pointer to a peer p_2. Alternatively, the item x could be *replicated* at p_2, both adding redundancy and allowing p_1 to control how much of the load to shed, although this then introduces issues of consistency. Again, in the case where items are placed using multiple choices, load shedding can attempt to maintain the invariants of the hashing paradigm, although this is not strictly necessary.

An interesting alternative combining replication and the multiple-choice schemes above is to replicate an item x at the k least loaded out of d possible locations given by hash functions. Such replication can maintain balanced load while also allowing additional functionality, such as parallel downloading from multiple sources [2, 3]. Indeed, parallel downloading may further improve load balancing in the system. This remains an interesting possibility for future study.

5 Experiments

In this section, we detail the results of our experiments. For comparison with the experiments of [10], we use 10^4 peers with numbers of items ranging from 10^5 to 10^6. The three schemes we consider use 1) $\lfloor \log_2 n \rfloor$ virtual peers, 2) an unbounded number of virtual peers (simulated using uniformly sized arcs), and 3) our power of two choices scheme ($d = 2$, breaking ties to smaller bins), respectively. We omit the basic scheme without any load balancing, since the virtual node scheme described in Chord [10] and in the introduction is clearly far superior.[3] All statistics are the results of aggregating 10^4 trials.

Figure 1 shows the 1st and 99th percentile loads for comparison to the results of [10]. Figure 2 shows the minimum and maximum loads – we view the maximum load as a key metric since the highest loaded peers are most likely to fail or provide poor service. Moreover, when a highly loaded node fails, its load cascades to its neighbors, which can then cause subsequent failures. Both figures show the mean load to illustrate how far or close each scheme is to the ideal. Figure 1(a) reproduces some of those

[3]The high loads that result when no load balancing is used (with 10^6 items and 10^4 peers, the 99th percentile load was 463 items per bin and the maximum load was 1820 items per bin) dwarf those of the three schemes we compare and make them difficult to distinguish when plotted on the same axes.

experiments of [10].[4] As noted there, the use of virtual peers improves load balancing significantly and reduces the fraction of idle peers compared to a scheme without load balancing. However, the corresponding maximum loads shown in Figure 2(a) are much higher and reveal a potential performance problem. Figures 1(b) and 2(b) show the results of using an unbounded number of virtual peers. The load balancing is significantly better in this case, but the maximum load is very similar to that shown in Figures 1(c) and 2(c), which show the benefits of employing two choices.

Overall, this means that even given unlimited resources to allocate to virtual peers in this scenario, the end result is a maximum load like that of using two choices. The distribution of load is slightly different – there is less variation in load than when using two choices – but we emphasize that we are comparing an unlimited resource scenario with a limited one. In particular, approximating the unlimited scenario is expensive, and the use of $\lfloor \log_2 n \rfloor$ virtual peers as proposed in [10] introduces a large amount of topology maintenance traffic but does not provide a very close approximation. Finally, we observe that while we are illustrating the most powerful instantiation of virtual peers, we are comparing it to the weakest choice model – further improvements are available to us just by increasing d to 3.

6 Conclusion

We advocate generalizing DHT's to enable a key to map to a set of d possible peers, rather than to a single peer. Use of this "power of two choices" paradigm facilitates demonstrably better load-balancing behavior than the virtual peers scheme originally proposed in Chord; moreover, it does so with considerably less shared routing information stored at each peer. We also make a preliminary case for other benefits of multiple storage options for each key ranging from fault-tolerance to better performance in highly dynamic environments.

At first glance, the prospect of having keys map to a small set of possible peers in a DHT runs the risk of incurring a substantial performance penalty. In practice, the cost is only a modest amount of extra static storage at each peer as well as a small additive constant in search lengths.

[4]The maximum load of a peer was not considered in [10].

References

[1] Azar, Y., Broder, A., Karlin, A., and Upfal, E. Balanced allocations. *SIAM Journal on Computing 29*, 1 (1999), 180–200.

[2] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed content delivery across adaptive overlay networks. In *SIGCOMM* (2002), pp. 47–60.

[3] Byers, J. W., Luby, M., and Mitzenmacher, M. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *INFOCOM (1)* (1999), pp. 275–283.

[4] Clark, D. The design philosophy of the DARPA internet protocols. In *ACM SIGCOMM* (1988), ACM Press, pp. 106–114.

[5] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Chateau Lake Louise, Banff, Canada, Oct. 2001).

[6] Karger, D. R., Lehman, E., Leighton, F. T., Panigrahy, R., Levine, M. S., and Lewin, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[7] Mitzenmacher, M., Richa, A., and Sitaraman, R. *The Power of Two Choices: A Survey of Techniques and Results*. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 255–312. edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim.

[8] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content addressable network. In *ACM SIGCOMM* (2001), pp. 161–172.

[9] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware 2001* (2001).

[10] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM* (2001), pp. 149–160.

[11] Vöcking, B. How asymmetry helps load balancing. In *Proceedings of the 40^{th} IEEE-FOCS* (1999), pp. 131–140.

[12] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, Apr. 2001.