

Cluster-based Optimizations for Distributed Hash Tables

Jeffrey Considine
jconsidi@cs.bu.edu
Computer Science Department
Boston University

November 1, 2002

Abstract

We consider the problem of performing topological optimizations of distributed hash tables. Such hash tables include Chord and Tapestry and are a popular building block for distributed applications. Optimizing topologies over one dimensional hash spaces is particularly difficult as the higher dimensionality of the underlying network makes close fits unlikely. Instead, current schemes are limited to heuristically performing local optimizations finding the best of small random set of peers. We propose a new class of topology optimizations based on the existence of clusters of close overlay members within the underlying network. By constructing additional overlays for each cluster, a significant portion of the search procedure can be performed within the local cluster with a corresponding reduction in the search time. Finally, we discuss the effects of these additional overlays on spatial locality and other load balancing schemes.

1 Introduction

Distributed hash tables (DHTs) have seen a recent surge in interest as a key building block in completely decentralized and distributed applications.

Distributed data structures bearing the name “distributed hash table” have been known for many years [5]. However, it has been the new wave of DHTs (Tapestry [10], Chord [9], CAN [7], etc) which has been the focus of new interest in the research community. These DHTs (and their derivatives) are de-

sirable since they all have decentralization, scalability, robustness and short overlay level paths. In particular, these schemes are scalable because each node need only be aware of and communicate with a small number of peers. Simultaneously, they still guarantee that the node responsible for some item can be found in a small bounded number of steps.

The DHTs we are interested in follow the general approach of Chord and map a one dimensional hash space onto the unit circle (the “Chord ring”). Both nodes and items are hashed onto the ring and items are assigned to the node immediately preceding them (counter-clockwise). By connecting each node to its successor, routing is possible, albeit slowly, by a simple clockwise traversal of the ring. In Chord, routing is sped up with a logarithmic sized “finger table”. The first “finger pointer” points to the node responsible for $1/2$ of the way around the ring, the second points $1/4$ of the way around the ring, etc. Routing then proceeds by repeatedly following the longest pointers not passing the desired item. This takes $O(\log n)$ hops with high probability. Derivative schemes such as [6, 1] achieve similar $O(\log n)$ hop routing using fewer but more elaborately arranged pointers but the general ideas are the same.

Relatively little work has been done with topological optimizations of Chord with the exception of [2]. In short, they note that the distance covered by a finger pointer is more important than the node it points to. Allowing approximations of the distance can allow significant flexibility in choosing which node to point to. Thus, the specified hash value is only used

as a target and the nodes close to it are all considered where previously only the node immediately preceding it was considered. Since latency is the key metric determining search time, each node in the neighborhood is probed and the lowest latency node is used for the finger pointer. It is reported that this optimization is very successful in practice.

Despite the practical success of this local heuristic, we note that it is unlikely to actually find the closest nodes on the network. The main reason for this is that there are only a few finger pointers from each node and correspondingly only a few nodes near them on the ring. The random placement of nodes on the ring makes it improbable that the closest nodes will be in the few spots close to the finger pointers. Additionally, we note that even knowledge of the underlying topology does not dramatically simplify the problem as this becomes a problem of efficiently projecting a higher dimensional space onto the ring. Therefore, instead of attempting to warp the underlying topology to match the ring or vice versa and possibly jeopardizing other desirable properties such as robustness, we propose the use of separate overlays to leverage topology information.

In brief, we first assume the existence of clusters of nodes in the network which are close to each other latency-wise. We start with a “full overlay”, a Chord ring composed of all of the nodes in the network. For each of the clusters, we construct an additional overlay called a “cluster overlay”. This cluster overlay uses the same hash functions as the full overlay so it forms a smaller Chord ring over the same hash space. Since the same hash function is used, a node has the same position in the cluster overlay’s ring and the full overlay’s ring. Therefore, given two nodes in the cluster overlay, one could traverse either overlay to move from one to the other. The main semantic difference in choosing between the two overlays is that the full overlay has more nodes so it will generally have nodes closer to any point in the hash space. Given a node in a cluster wishing to perform a search, the following method is suggested. First, the searching node initiates a search within the cluster overlay and finds the node responsible for the desired item. Then, the searching node continues its search within the full overlay to find the

desired item. The first portion of this search should have a significant per-hop speedup since it is entirely within a cluster. For reasonably sized clusters with a very fast local network, the improvement can be quite dramatic. See Figure 1 for an example.

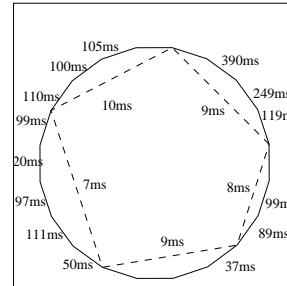


Figure 1: Full overlay vs cluster overlay (without finger pointers)

Example 1 Consider a popular file sharing program which has about a million users online at any point, of which a thousand are at a particular large university campus. From this university, the average RTT to an outside destination is 100ms while the average RTT to a local host is 10ms. Using a scheme similar to Chord, searching takes approximately 20 steps, so the expected search time is 2000ms. By adding a cluster overlay covering the campus network, these steps will be split into about 10 in the campus network and 10 in the rest of the network. In this case, the expected search time is 1100ms, almost a factor of two improvement.

The use of cluster overlays as described so far is surprisingly simple but can be very effective. Before dwelling on the details, we first attempt to give some more intuition into how they work. One view of cluster overlays is that they provide fast but low resolution coverage of the hash space allowing one to quickly zoom in on the right area of the full overlay. An alternative view considers the tree of search paths leaving a node and notes that all of the upper portions of the tree (near the root) are now within the cluster. Yet another puts it forth as an answer to the question, “How close can we get to our destination without leaving the cluster?”.

We elaborate on the details of using cluster overlays in Section 2. Extensions to cluster overlays which improve the number of hops searching and the spatial locality of query responses are then considered in Section 3. We conclude in Section 4 relating our work to other schemes and expanding on directions for future work.

2 Cluster Overlay Details

In this section, we delve into the details and analysis of cluster overlays.

For simplicity, we only consider “iterative” implementations where each node is in at most one cluster. That is, searches are performed entirely by the originating node which contacts each node of the traversal in turn. We use n and n_c to denote the number of nodes in the full overlay and cluster overlays, respectively. Similarly, RTT and RTT_c denote the average round trip times to nodes in the full and cluster overlays from the node performing the search.

First, we consider the speedups in search time using cluster overlays, since they are the primary motivation of this work. The characteristic number of hops performing a search in a Chord overlay is $\Theta(\lg n)$, where the constant factors vary according to assumptions about uniform request distributions and whether a high probability result or just an average case result is desired. We simplify these bounds to the formula $\alpha \lg n$ which describes the typical number of hops under Chord within an additive constant, for $\alpha = 1/2$. Searches over the full overlay then take an expected time $RTT * \alpha \lg n$. When using cluster overlays, it is straightforward to argue that the (simplified) expected time is $RTT_c * \alpha \lg n + RTT * \alpha \lg(n/n_c)$. This gives a speedup of $\frac{RTT \lg n}{RTT_c \lg n_c + RTT \lg n - RTT \lg n_c}$. As RTT_c decreases, this approaches a limit of $\frac{\lg n}{\lg n - \lg n_c}$. If we express n_c as a power law in terms of n , i.e. $n_c = n^x$, then the limit of the speedup is $\frac{1}{1-x}$ - it is entirely determined by the power law exponent. That is to say, if $n_c = n^{1/2}$, then a speedup of up to 2 is possible. If $n_c = n^{1/4}$, then the speedup is at most 4/3.

Next, we consider the overhead in edges from

adding an additional overlay. This metric is important since each edge in the overlay will need to be probed at regular intervals as part of a maintenance process. In Chord, the default configuration is for each node to maintain $\lg n$ finger table entries and $\lg n$ successor pointers to the nodes immediately following it in the ring. With this configuration, each node in a cluster overlay of n_c nodes will need to maintain an additional $\lg n_c$ successor pointers. For the successor pointers however, with the exception of the immediate ring successor, their primary purpose is to maintain the ring in case of failures. Keeping this in mind, only one successor pointer in the cluster overlay is strictly necessary - failure cases can be handled by rejoining through the full overlay which does survive with high probability (see Appendix A for details). Thus, when adding a cluster overlay to a Chord ring using $2 \lg n$ edges, only $\lg n_c + 1$ additional edges are necessary to support the cluster overlay - one for the cluster overlay successor and $\lg n_c$ for the cluster overlay’s finger table. Therefore, the number of edges to support the cluster overlay is no more than an additional 50% beyond those for the full overlay. Additionally, we note that higher overheads here correspond to higher speedups since more hops along a search path will be within the cluster.

Initialization of the extra finger table entries is also surprisingly cheap if the cluster overlay is joined first. Once a node in the cluster overlay is found, the joining node performs a search from the found node to the joining node’s position on the ring. The important observation here is that the point on the cluster overlay where this search switches to the full overlay is the new nodes position in the cluster overlay. Similarly, when initializing the full overlay finger tables, the cluster overlay nodes where the searches switch are the corresponding entries of the cluster overlay finger tables. This makes the only overhead of joining a cluster overlay the initial search to find a cluster overlay node - the remainder of the initialization is expected to be faster since it can leverage the cluster overlay.

In summary, the primary overhead of maintaining a cluster overlay is in the initial bootstrapping to find it and maintenance of the extra edges - all other costs are amortized away as part of optimized searches for maintaining the full network. As a final note with respect to

these costs, if they are still deemed excessive, then the idea of cluster overlays applies equally well to fixed degree schemes such as [6, 1].

3 Extensions

In this section, we discuss two more classes of optimizations that are possible using this framework. The first takes advantage of the cluster overlay to distribute knowledge of all nodes known to the cluster. The second strengthens spatial locality by considering the effects of separately indexing content within the cluster.

3.1 Node Indexing

Using the search method presented earlier, the cluster overlay is traversed until the last cluster node before the desired item is found, at which point the search continues along the full overlay. However, this is not the closest point on the full overlay known to the cluster as a whole. That is, the cluster nodes' finger tables for the full overlay may have pointers to full overlay nodes between the closest cluster node and the desired item.

The collective finger tables of the cluster nodes can be leveraged by also inserting them into the cluster's distributed hash table. Each node then is responsible for knowing about any full overlay nodes that lie within its domain of responsibility. The search method is then modified to find the closest node in the cluster, query it for the closest full overlay node it is aware of, and continue the search from that node.

The average number of nodes stored at each node in this fashion will be $O(\log n_c)$, thus saving $O(\log \log n_c)$ hops. Maintaining this information may seem expensive compared to the small benefits incurred, but we note that it can be done very cheaply with a simple observation relating the full and cluster finger tables. That is, the i th entry of the full finger table always falls in the domain of the i th entry of the cluster finger table so it can be trivially integrated with the update process. More generally, any nodes traversed after leaving the cluster overlay could be stored, but we do not explore this avenue.

3.2 Separate Indexing

So far, we have only discussed the use of cluster overlays to speedup searches. Another natural application is to provide spatial locality for search results. For many applications such as file transfers, spatial locality can be helpful, both in finding faster sources and in limiting traffic to the local network, thus saving more costly outgoing links.

Spatial locality in searches can be easily provided if each node also maintains a second index of items held by cluster nodes. Each node will be responsible for a larger range of the hash space since there are fewer cluster nodes, but there are also fewer nodes with items to index. As noted in Section 2, searches for the responsible node in the full overlay will pass through the responsible node in the cluster overlay, so updates will essentially come for free unless the items are large.

In a similar vein, we note that cluster overlays interact very nicely with the caching scheme of DHash and each cluster need only retrieve an item externally once before always retrieving a local copy (or a logarithmic number of times under their proposed refinement).

4 Conclusions and Future Work

We have presented a new approach to performing topological optimizations of distributed hash tables. While our presentation has focused on Chord overlays, our methods naturally work with other overlays such as Tapestry and to a lesser extent, higher dimensional ones such as CAN. Similarly, there are straight-forward extensions to hierarchies of clusters by starting within the smallest and fastest clusters and gradually moving out through the slower clusters. We plan to both explore these extensions and test the effectiveness of cluster overlays in real environments.

Acknowledgments

We are grateful to John Byers, Marwan Fayed, Thomas Florio and Anukool Lakhina for thoughtful discussion during the development of these ideas.

References

- [1] CONSIDINE, J., AND FLORIO, T. Scalable peer-to-peer indexing with constant state. Tech. rep., CS Department, Boston University, September 2002.
- [2] 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).
- [3] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences* 31, 2 (1985), 182–209.
- [4] GIBBONS, P. B., AND TIRTHAPURA, S. Estimating simple functions on the union of data streams. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures* (2001), ACM Press, pp. 281–291.
- [5] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH* a scalable, distributed data structure. *ACM Transactions on Database Systems (TODS)* 21, 4 (1996), 480–525.
- [6] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing* (2002).
- [7] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content addressable network. Tech. Rep. TR-00-010, Berkeley, CA, 2000.
- [8] RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. Topologically-aware overlay construction and server selection. In *Proceedings of IEEE INFOCOM'02* (6 2002).
- [9] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.
- [10] 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.

A Infrastructure Support

Throughout this discussion, we have assumed the existence of clusters and the ability to identify nodes within the cluster to construct the cluster overlay. While we wish to keep the methods of cluster discovery orthogonal to our presentation, it is important to be able to test for the existence of a cluster overlay, estimate its size (or the number of potential members) and randomly sample its members for bootstrapping purposes. Some methods of cluster discovery may provide some of this information implicitly (e.g. any based on extensive measurements of the actual topology). Others may only identify potential clusters without indicating whether any other nodes lie within them (e.g. simple methods based on IP-prefixes, DNS suffixes, originating AS or geographic location or more involved ones such as beaconing [8]). Regardless, we sketch how to build support for these operations over the full Chord ring so as to avoid the need for any centralized resources to support cluster overlays.

Approximate counting techniques for summarizing streams originated with [3]. The main goal of these techniques is to construct a concise summary that allows the number of distinct items in the stream to be estimated. Generally, these summaries also have a cheap union operation allowing the summaries of individual streams to be combined to form a summary for their union. By using the same search tree structure leveraged by DHash and using the provided union operation to combine children, the total number of cluster members can be cheaply estimated, even as members join and leave changing the tree structure. To extend this structure to random sampling, one need only use the summaries of [4] which include a sample of the nodes in their stream. Sampling can then be performed by starting at a random location on the ring, and from there searching for the root of the tree and taking a node from the first summary encountered.