CS268 Course Project

Understanding Chord Performance

and Topology-aware Overlay Construction for Chord

Li Zhuang(zl@cs), Feng Zhou(zf@cs)

Abstract

We studied performance of the Chord scalable lookup system with several performance enhancements. For the denser finger technique, we gave analytical result that shows the average lookup path length decreases slower than the maximum length and converges to it as finger density grows higher. Location cache, as implemented in MIT Chord, is shown to be able to reduce the lookup path length by 1/2 of the logarithm of cache size, which makes it very effective in static networks. However, our experiments with the MIT Chord implementation shows that because of the stale-entry problem, the location cache does not scale to more than 2000 nodes in a typical file-swapping network setting and there are plenty of room for improvement. We found that server selection is an effective way of reducing lookup stretch. In the ideal exponential delay network, Chord with server selection and a $O(log^2 N)$ entry routing table will have $O(1)$ lookup stretch. In a transit-stubs network, server selection still achieves 50% improvement of lookup stretch. The other part of our work is a topology-aware overlay construction scheme for Chord. During the setup of the network, Principle Component Analysis is used to select representative landmarks from a pool of nodes and place the landmarks to appropriate positions on the Chord ID circle. Later a two-level approach is used to assign each new node a locally optimal ID. Preliminary results show that this improves routing locality of the Chord algorithm. 21% reduction in lookup stretch is observed in a 600-node transit stub topology.

1 Motivation

Structured overlay networks [10], [6], [4], [9] provides similar $O(\log N)$ bounds on lookup time measured in hop counts. This asymptotic lookup path length, combined with low per-node state, guarantees the scalability of such systems. However, other factors such as the constants in $O(\log N)$ and average latency of each overlay hop, affects the performance of these systems significantly. A plethora of techniques have been proposed to enhance lookup performance, including proximity routing ([1], [4] and server selection [2]), geographic overlay construction [6] and landmark routing [12]. In the case of Chord[10], the classic algorithm is simple and has provable correctness and robustness. But its lookup performance is relatively low compared to Tapestry[4], Pastry[9] and others exploiting routing locality. There have been discussion about possible improvements and the MIT Chord has implemented some of them. However, few results are published on these improvements. The first part of this report presents analysis and experiment results of several existing performance enhancing techniques of Chord. The purpose of our work is to find out how Chord with these improvements perform in large networks and changing networks. Our methodology is a combination of analysis, simulation and benchmarking.

The second part proposes a topology-aware overlay construction scheme for Chord. It assigns similar IDs to nearby nodes in the network. Because most overlay hops during lookup only skip small portions of the whole ID circle, constructing the overlay this way will improve routing locality. A fundamental question [7] here is how much we can possibly improve, especially when the ID space is one dimensional in Chord. Instead of trying to pursue the question theoretically, we try to answer it by proposing an actual scheme and doing experiments. This is on-going work and we present preliminary results.

2 Chord Performance, with Existing Enhancements

2.1 Average Lookup Path Length with Denser Fingers

The Chord protocol as presented in [10] keeps very little per-node state, compared to other systems such as Pastry and Tapestry. Only $\log N$ fingers, one successor and one predecessor pointers are kept per node. $successor(id + 2^n)$ are kept as *fingers*, which are overlay routing table entries. Fingers are followed during routing to find the owner (successor) of the target id. In future work part [11], the authors propose a generalization to Chord, using denser fingers as a performance enhancement. Now fingers are placed at points $successor(id + (1 + 1/d)^n)$. d is a tunable integer parameter. A larger d results in denser fingers and $d = 1$ corresponds to the normal Chord. Under this scheme, a routing hop will decrease the distance to query target to at most $1/(1 + d)$ of the original distance. Therefore assuming even node distribution, the query target will be found in at most $\log_{1+d} N$ hops, which is $1/\log(1 + d)$ of the original max path length. The number of fingers kept by each node is now $\log N/\log(1 + 1/d) \approx O(d \log N)$. By keeping d times fingers, maximum path length is reduced to $1/\log(1 + d)$ of the original.

However, average path length, not the maximum, is often more interesting to us. Below we give an estimation of the average path length of this scheme. It turns out that the improvement is not as good as for the maximum path length. First, we have the following property,

Property 1. Suppose the average query path length of a N node Chord network is $L_{avg}(N)$. Then the average path length of a lookup on a portion α $(0 < \alpha \leq 1)$ of the Chord ID circle is approximately $L_{avg}(\alpha N)$.

In other words, querying on a part of the Chord ID circle is the same as querying a whole Chord network with the same number of nodes. The correctness is easy to see. It is because at each hop during the lookup, the fingers divides the relevant portion of ID circle into pieces of same proportions. Therefore the number of hops is determined by how many level of fingers are there for each piece, which is in turned determined by the total number of nodes within the ID range.

Also observe that the farthest finger of any node divides the ID circle into two portions: $1/(1+d)$ for all IDs larger than the finger and $d/(1 + d)$ smaller. If the query target ID falls into the first $1/(1 + d)$, the first finger will be followed and that node will in turn handle the query. If it falls into the second part of length $d/(1 + d)$, it will be handled locally. Therefore using the previous property, we have this for average lookup path length $L_{avg}(N)$,

$$L_{avg}(N) = \frac{1}{1+d}(1 + L_{avg}(\frac{N}{1+d})) + \frac{d}{1+d}L_{avg}(\frac{dN}{1+d})$$

We know $L_{avg}(N) = c_{avg} \log N$ asymptotically, where c_{avg} is a constant that depends on d. Therefore by solving this equation for c_{avg}, we get,

$$L_{avg}(N) = \frac{1}{(1 + d)\log(1 + d) - d \log d} \log N$$

When $d = 1$, i.e. normal Chord, we have $L_{avg}(N) = 0.5 \log N$, which agrees with experiments in [11]. The maximum path length, $L_{max}(N)$, is $\frac{1}{\log(1+d)} \log N$. Figure 1 shows the relationship between $L_{avg}(N)$ and $L_{max}(N)$. For example, when $d = 32$, $L_{avg}(N) \approx 0.8 L_{max}(N)$. It shows the improvements we get by increasing d is less than that suggested by the trend of $L_{max}(N)$. Actually, when d is large,

$$L_{avg}(N) = \frac{1}{\log(1 + d) + d \log(1 + 1/d)} \log N \rightarrow \frac{1}{1 + \log(1 + d)} \log N \rightarrow L_{max}(N)$$

Figure 2 is the result obtained from simulation. The network size is 10,000 nodes and 10,000 lookups from random nodes are done for each d value. It confirms the analysis result.

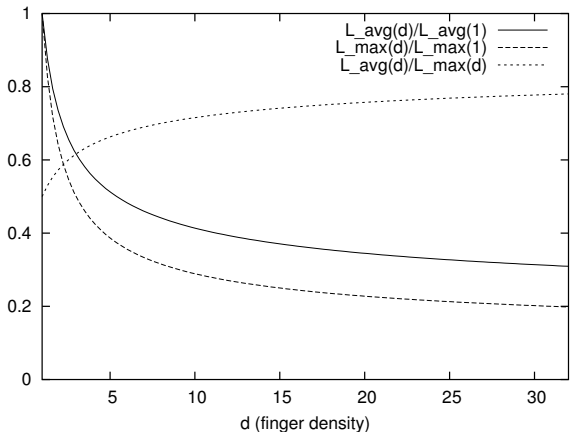

Figure 1: Average lookup path length vs. maximum with different finger density

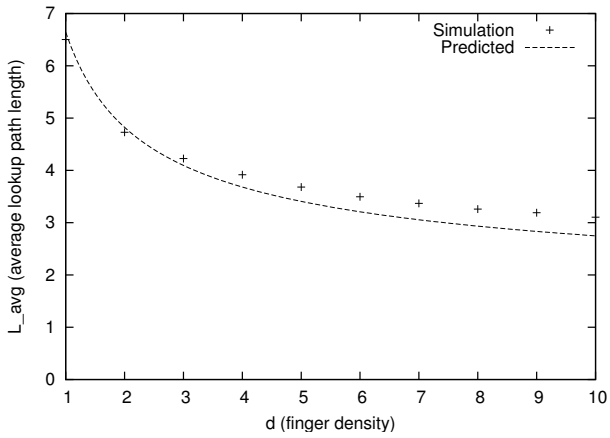

Figure 2: Simulated and predicted path length with different finger density

In summary, denser fingers decrease lookup path length. But the improvement in average case is slightly less than previous conjectured.

2.2 Location Caching

The MIT Chord implementation [5] includes a often overlooked performance feature, a location cache that remembers most recently seen ID-to-host mappings. During lookup operations, location cache entries are treated the same as fingers (actually fingers are implemented simply as *pinned* location cache entries). Benchmark results with the location cache (in original version of [8]) has shown that in a small network (83 nodes), this cached is very effective and Chord out-performs Tapestry, which uses proximity routing, by 37%. In contrast, the revised version of [8] shows that without the cache, Chord is 2.9 times slower than Tapestry. Actually, if the predecessor node of the target ID is in the location cache of the querying host, the query path length is always 1. For lookup-intensive applications like CFS [2], this will greatly increase performance and reduce total traffic. However, despite the obvious improvement the location cache brought by, there has been little discussion about it. Most papers simply left it as an implementation detail. This contrasts with the emphasis placed on caching in other topics such as collaborate Web proxies and operating systems in general. Therefore we believe it is interesting to find out the benefits and limitations of this technique.

2.2.1 Estimation of cache effectiveness

For a very large network, certainly nodes can not cache location of every other node. But this per-node state limit is not a real limit. Each location cache entry in Chord takes 72 bytes. Thus a 10 MB Chord location cache has about 150,000 entries, which should be enough for most applications. Even if locations of only a small portion of hosts are cached, significant improvement can be achieved. Let C be the number of hosts cached. Intuitively, evenly distributed cache entries divides the Chord ID circle into segments of same length. Therefore the first hop will jump into one of these segments and the number of hops needed later is equal to lookup path length in a N/C-node network. The expected lookup path length is roughly,

$$\bar{L_c}(N, C) \approx \frac{1}{2} \log \frac{N}{C} + 1 = \bar{L}(N) - \frac{1}{2} \log C$$

This shows a unique feature of the location cache.

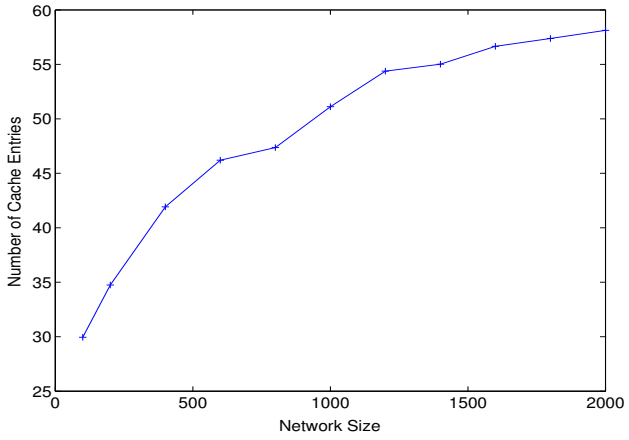

Figure 3: Average number of location cache entries of a new node after stabilization

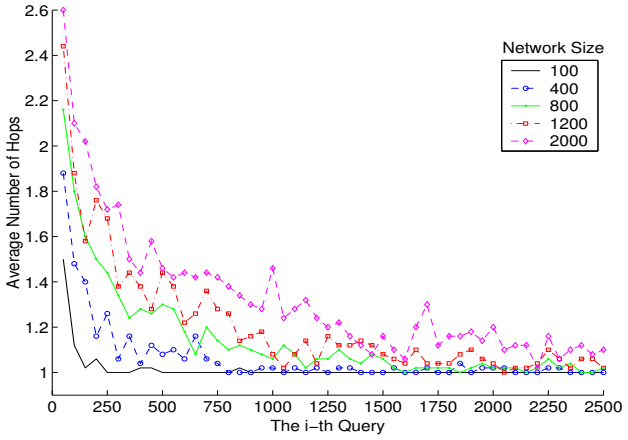

Figure 4: Changing average query path lengths over time with location cache in a stable network

Property 2. The number of query hops saved by location cache is logarithmic to the number of cache entries. In other words, the first few cache entries are most effective in reducing path length.

For example, in a 4,000-node network, caching only 64 nodes (1.5%) will halve the lookup path length, from 6 to 3. An interesting question is whether this nice feature is a Chord privilege. It is not clear how this large a reduction in query path length can be achieve in Pastry or Tapestry with only a few cache entries.

Moreover, another significant benefit of location caching is that cache entries are obtained "for free". Unlike denser fingers and server selection (discussed later), the node doing lookup does not need to send out packets to obtained the entries. They are accumulated for free.

2.2.2 Cold cache behavior

When a new node joins the network, it has a cold location cache. So we expect it to have much longer query path than other nodes. However this turns out to be not true. When a node has joined the network, it already has quite a few cache entries as a result of the stabilization (idealization) process. As a result of property 2, this will significantly reduces its lookup path length right from the first lookup. Figure 3 shows the average number of cache entries of a new node after first stabilization, for different network sizes. This is obtained from actually running that number of Chord instances.

Figure 4 shows the average query path lengths of performing a series of random queries on a particular node. Different lines show results for different network sizes. No other queries are going on on other nodes during the process. The experiment is done in a cluster, by running many Chord nodes on each physical node. Because the network is static with no nodes joining and leaving, the LAN environment has on effect on the average query path length. The length of the first queries are significantly lower than without cache, e.g., for a 2000-node network, the first query takes 2.6 hops, compared to 5.319 hops without cache. The path lengths drop quickly as more queries are done and more nodes are cached. Lengths for all network sizes fall below 1.5 after 500 queries. They converge to 1, the absolute minimum, as nearly all nodes become cached. Figure 5 shows the result of doing lookups from random nodes.

2.2.3 Caching in a changing network

The major limitation we found in the current location cache of MIT Chord implementation is it does not deal with changing networks well. The main reason is that because cache entry validity is not actively

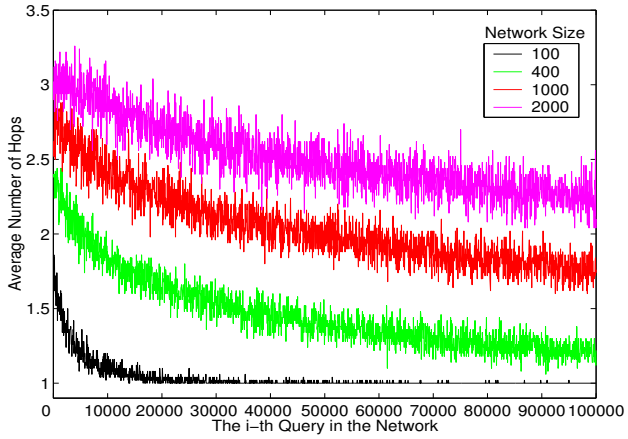

Figure 5: Changing average query path lengths over time with location cache in a stable network (each lookup done from a random node)

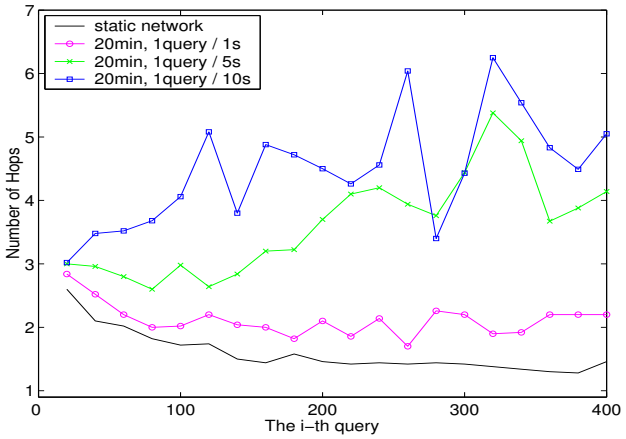

Figure 6: Average query path lengths in a 2000-node dynamic network with average node life of 20 minutes

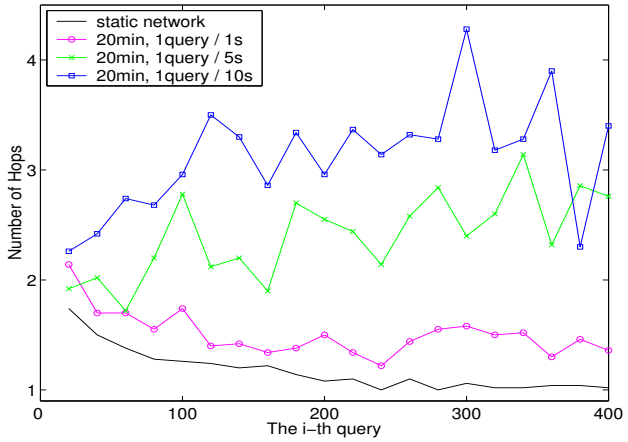

Figure 7: Average query path lengths in a 400-node dynamic network with average node life of 20 minutes

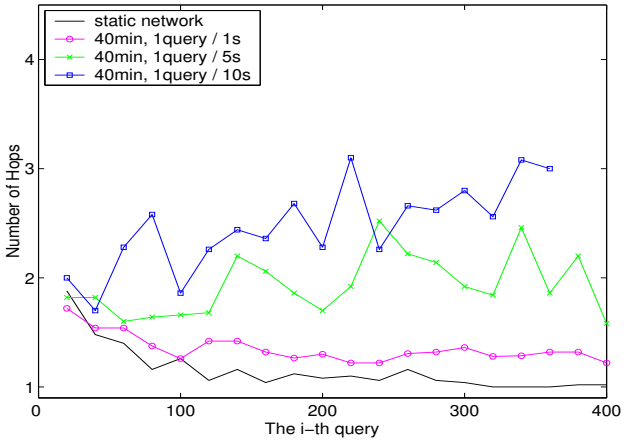

Figure 8: Average query path lengths in a 400-node dynamic network with average node life of 40 minutes

verified, a node tends to have more and more stale cache entries over time. This results in time-outs and redundant hops. In contrast, fingers are periodically verified, although in a lower frequency than successors. Therefore in a highly dynamic network, routing using solely fingers could be better than using location cache.

Figure 7 and 8 compares average path lengths in the same 400-node network, with 20-minute average node life time and 40-minute average node life time, respectively. This experiment is also done in the LAN environment. Different lines represent different query frequencies. Nodes doing more queries per minute get lower query path lengths because each query refreshes part of the location cache. Queries in the faster-changing network (figure 7) have longer path length than the slower one. Additionally, figure 6 shows the same 20-minute life time experiment for a 2000 node network. This one has higher (by 2) pikes than the 400-node case. In a larger network, with the same query speed and node life time, refresh interval of each cache entry becomes longer. This is one scalability limit of the location cache.

These two figures give a rough idea of how dynamic the network can be for the cache to remain useful. The average routing path length using fingers of a 2000-node network is 5.32 (obtained from simulation), assuming fingers are maintained often enough to remain 100% valid. When the average

node life time is 20 minutes, a node doing 1 query per 10 seconds get nearly the same average query path length using location cache and using fingers. This basically says the location cache will not save network resource in this scenario. Moreover, lookup latency with the cache will be a lot higher because of communication timeouts.

In summary, we see this stale cache problem as a *real limitation* of the location cache. For quickly-evolving networks like a file-swapping peer-to-peer network, in which 20 minutes life time and 0.1 per second query frequency are reasonable estimations, the current location cache in Chord will not scale to 2000 nodes. Nevertheless, the location cache will still be a very useful technique in a lot of applications in which networks are changing more slowly.

2.3 Server Selection

Server selection is a kind of proximity routing, proposed in [2] and [11] to improve Chord lookup performance. The idea is select the next hop to be not always the farthest non-overshooting finger, but some node that is nearer in ID space but also nearer in terms of network latency. The hope is to avoid jumping to nodes that are far away from both the starting node and the destination node.

There are different server selection schemes. The server selection scheme in CFS [2] (referred to as *inter-finger server selection* hereafter) is based on query path length estimation. The number of ones in the first $\log N$ bits of the ID-space distance between a finger and the query ID gives an estimation of the length of the remaining query path from that finger. Multiply that estimated hop count by the average hop latency, plus latency to that finger will yield an estimation of the total query latency from current node if that finger is taken. The server selection algorithm selects the finger with the smallest estimated latency.

The other server selection scheme proposed [11] (referred to as *intra-finger server selection* hereafter) is to keep pointers to k consecutive nodes for each finger entry, all in that finger's ID range. The k nodes are almost equivalent for routing on the ID circle. But their network latency from the current node can be very different. Therefore the lookup can be forwarded to the node with lowest latency. An interesting property we found about this scheme is,

Property 3. Assuming exponential random latency between nodes and recursive lookup style, a Chord network can achieve O(1) stretch lookup regarding network size using intra-finger server selection. The per-node state needed is $O(\log^2 N)$.

Proof. Suppose k pointers are kept for each finger. These nodes are random in the actual network. Let their latency from the current node be D_i $(1 \le i \le k)$. $\{D_i\}$ are all independent random variables with exponential distribution. Let the expectation of D_i be d. Then the expected server-selected per-hop latency is,

$$d_{ss} = \text{Min}\{D_i\} = d/k$$

This is the result of a well-known property of exponential distribution: the minimum of k independent exponential random variables is also exponential distributed, with parameter $k\lambda$.

This server selection procedure goes on until the second last hop, where the predecessor of the ID is reached. Thus expected latency of the second last hop is d. The last hop goes from the predecessor to the starting node, also with expected latency of d. Suppose server selection does not change query path length (actually it will be a little shorter), the total expected query latency is,

$$S = (L(N) - 1)d_{ss} + 2d = (\frac{1}{2}\log N - 1)\frac{d}{k} + 2d$$

Stretch of the query is,

$$s = \frac{S}{2d} = \frac{\log N - 2}{4k} + 1$$

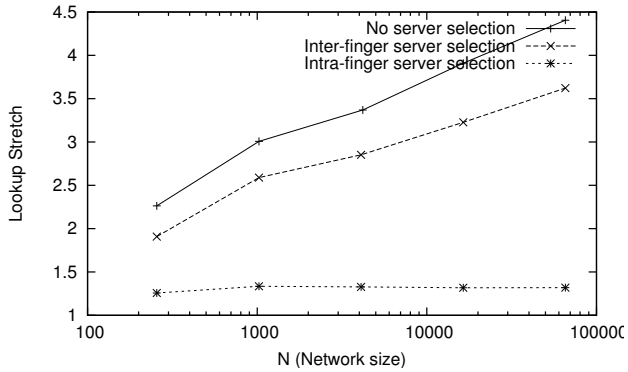

Figure 9: Stretch in exponential delay networks

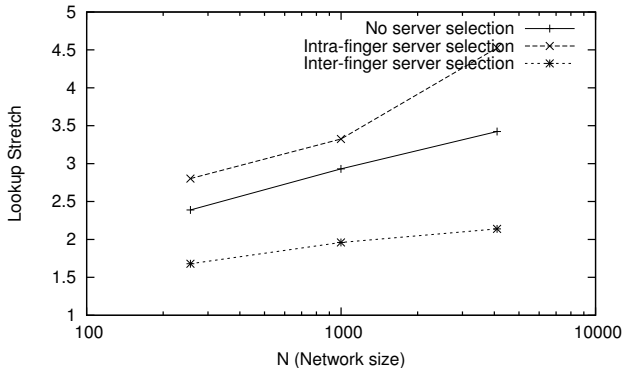

Figure 10: Stretch in transit-stub networks

Setting $k = c \log N$ will give us,

$$s = 1 + \frac{1}{4c} - \frac{1}{4c \log N} = O(1)$$

■

s converges to $1 + \frac{1}{4c}$ when N gets to infinity. When $c = 0.5$, the upper bound of stretch is 1.5.

Figure 9 shows simulation results for exponential delay networks, with no server selection, with inter-finger server selection (CFS server selection) and with $\frac{1}{2} \log N$-node-per-finger intra-finger server selection (i.e. $c = \frac{1}{2}$). The results agree with the above analysis. Stretch of intra-node server selection remains below 1.5, while with no server selection or inter-node server selection, it goes up logarithmically with network size.

The exponential delay assumption is not realistic. Figure 10 shows results of the same experiment on transit-stub networks. Still, intra-finger server selection has much lower stretch than no server selection and inter-finger server selection. One surprising result is inter-finger server selection actually increases stretch in this topology. It is probably due to large errors in the query path length estimation function. Counting number of ones is not a perfect estimation, because actual fingers are not exactly at 2^n positions. They are *after* these position. So each finger can change more than one bit of the distance. In another experiment (detailed result omitted), we found that using just logarithm of the distance as path length estimation gives much better result, although it still improve stretch by no more than 5% compared with no server selection.

In summary, by keeping more state ($O(\log^2 N)$) per node, Chord can achieve $O(1)$ stretch in exponential delay networks using server selection. The stretch for real networks is higher but still much better than without server selection. Note that in practice, the constants in this $O(\log^2 N)$ can be quite small ($\frac{1}{2} \log^2 N$ in our experiments). In comparison, Tapestry and Pastry has $O(\log N)$ entries in their routing table. But it is indeed $b(\log N / \log b) + c$, where b is the "radix", typically 16. For a lot of useful network sizes, this is not much smaller than $\frac{1}{2} \log^2 N$.

3 Topology-aware Overlay Construction in Chord

In current Chord design, the distance (i.e. latency) of each application hop is half of the average network diameter. Even through the average Chord path is $O(\log n)$, a Chord query of still traverses great actual network distance. An extreme example is that a Chord node queries objects right on its successor might traverse average network diameter!

Intuitively, we expect the distance in ID space represents the the actually network distance. It is obvious this property is desirable for every overlay structure network. For example, in Chord, we want to project the geographic network "topology" to an one-dimension Chord ID space.

In CAN, the landmark reordering approach is used [6]. In CAN, the d-dimension ID space is cut to $m!$ cells with i pieces at the i-th dimension. A node is put in to the cell corresponding to the order of RTTs to m landmarks. However, landmark reordering, which projects the network topology to a high dimension ID space, can not be used in Chord, Pastry and Tapestry, which all use one-dimension ID space. A topologically-aware construction method for overlay network which works for one-dimension ID space, will benefit Chord as well as Tapestry and Pastry.

Here we proposed a classification based two-level geographic overlay construction method applicable to any ID space.

3.1 Two Level landmarks

Initially, we have m bootstrap nodes randomly chosen from the whole Internet as "landmarks", say $m = 50$. Let D' represents the network distance matrix among these "landmarks":

$$D' = \{d'_{ij}\}_{i,j=1,\cdots,m}$$

where d'_{ij} is the network distance between node i and j.

Some "landmarks" are very near each other, and the distance vector of these "landmarks" are similar. Little extra geographic information we could get when a host pings more than one of these similar "landmarks". So, it is straight forward that we want to cluster the m "landmarks" based on the distance matrix, and find some "representative" ones, which we call "level-1 landmarks". Moreover, it is desirable if we could put the "level-1 landmarks" at the right position in ID space where matches its relative network location to all other landmarks.

This is a vector classification problem with known class specifications, which is suitable to be solved by the Principle Component Analysis (PCA) method. What PCA could do is analyzing the most important information for classification (which is called Principle Components, PCs) among the vectors, then projecting the vectors to the PCs and getting a lower dimension vector. The lower dimension vectors remain the classification-useful information and throw classification-useless information. So, by comparing distances between lower dimension vectors, we can more efficiently cluster the original vectors. We briefly describe it below, and the detailed description is available in any AI or digital image processing book (i.e.[3]).

Suppose we have random m-dimension vector population d, where $\vec{d} = (d_1, d_2, \cdots, d_m)^T$. We make n independent observations of d: $\vec{d}_1, \vec{d}_2, \cdots, \vec{d}_n$, and compute the mean of the population $\vec{d}$ as: $\vec{\mu}_d = E(\vec{d}) = \frac{1}{n}\sum_{i=1}^{n} \vec{d}_i$. So, the covariance matrix of the same data set is:

$$C_d = E\{(\vec{d} - \vec{\mu}_d)(\vec{d} - \vec{\mu}_d)^T\}$$

For symmetry matrix like covariance matrix, we can compute the eigen-vectors and eigen-values of matrix C_d based on the characteristic equation: $|C_d - \lambda I| = 0$. By ordering the eigen-vectors in the order of descending eigen-values (largest first), one can create an ordered orthogonal basis with the first eigen-vector having the direction of largest variance of the data. The eigen-values / eigen-vectors of covariance matrix have the feature that very small number of the eigen-values (i.e. 10%) add up to more than α% (i.e. 95%) of the sum of all eigen-values. And also the "large eigen-vector" (or a linear combination of several "large eigen-vectors") represents a typical cluster. We call the corresponding eigen-vectors of largest α% eigen-values ("large eigen-vectors") the Principle Components (PCs) of the random population $\vec{d}$. Suppose we select k PCs of C_d : $A = (\vec{c}_1, \vec{c}_2, \cdots, \vec{c}_k)^T$, then an observation of $\vec{d}$ can be represent as:

$$\vec{d} = A^T\vec{y} + \vec{\mu}_d, \text{ where } \vec{y} = A(\vec{d} - \vec{\mu}_d)$$

with little useful information loss. Compared with $\vec{d}$, $\vec{y}$ is a lower dimension vector which retains information useful for classification while removes useless information and noise in original $\vec{d}$. The distance between the $\vec{y}$'s can represent the distance between the $\vec{d}$'s.

Now, we illustrate how the PCA is applied to cluster the landmarks. We evenly cut the ID space into m bins, we can compute the distance matrix in ID space as:

$$D = \{d_{ij}\}_{i,j=1,\cdots,m}$$

Where d_{ij} is the ID space distance between i-th bin and j-th bin.

We do PCA on matrix D, then we get principle component matrix A. For each ID space distance vector $\vec{d_i}$ $(i = 1, \cdots, m)$, compute its lower dimension vector $\vec{y_i}$. For each network distance vector $\vec{d'}_j$ also compute its corresponding lower dimension vector $\vec{y'}_j$. We put network node j to the i_0-th bin in ID space, if

$$\|\vec{y}_{i_0}, \vec{y'}_j\| = \min_{i=1,\cdots,n} \|\vec{y_i}, \vec{y'}_j\|$$

where $\|\cdot\|$ represents the vector distance. We use the "angle" between two vectors as "vector distance".

Thus, we put the m network landmarks to the m bins in the ID space. Notice that some bins may have no landmarks while others may have more than one. It is easy to cluster landmarks in the nearby bins together using unsupervised learning method in AI. For example, we cluster m network landmarks into b clusters:

$$\{\{\text{landmark}_{i_{1,1}}, \text{landmark}_{i_{1,2}}, \cdots\}\{\text{landmark}_{i_{2,1}}, \text{landmark}_{i_{2,2}}, \cdots\} \cdots \{\text{landmark}_{i_{b,1}}, \text{landmark}_{i_{b,2}}, \cdots\}\}$$

where the landmarks above are ordered according to the bin where it is put, and each landmark is in charge of $1/m$ ID space.

From each cluster, we select one landmark as its representative "level-1 landmark". All landmarks in the same cluster is called a cluster of "level-2 landmarks". It is obvious RTTs between a host to "level-1 landmarks" represent its global location in the Internet. The "level-2 landmarks" in the same cluster is analogous to landmarks in the same AS. Once we know which cluster a host belongs to, the the RTT to "level-2 landmarks" in that cluster represents its local location in that AS.

We conclude the alignment of the initial m landmarks in the ID space have following features:

- Landmarks near each other are put in ID space near each other.
- For each landmark, its distance vector in actual network and its distance vector in ID space are well matched. Thus, its location in ID space is a good represent of its location in the actual network.
- The approach above can be applied in either 1-dimension ID space or multi-dimension ID space as long as the "distance" in ID space can be defined.

3.2 Compute ID for a Node

Suppose we have b "level-1 landmarks" and each cluster averagely has l "level-2 landmarks". When a new node wants to join the network, it first measures RTTs to the "level-1 landmarks" and get a network distance vector $\vec{X}_{L1} = \{x_1, \cdots, x_b\}$. Since we also know the distance matrix $L1$ of "level-1 landmarks", so we put this node in the b_0-th cluster, if

$$\|\vec{L1}_{b_0}, \vec{X}_{L1}\| = \min_{i=1,\cdots,b} \|\vec{L1}_i, \vec{X}_{L1}\|$$

With b_0, we get the b_0-th local cluster of "level-2 landmarks" with distance matrix $L2_{b_0}$. This node then measures its RTTs to all these "level-2 landmarks" and gets a vector $\vec{X}_{L2}$. Similarly to level-1, it computes its nearest local landmark l_0 and put itself at a position near l_0 in ID space. Suppose the ID space the l_0-th "level-2 landmark" in charge of is $[ID_1, ID_2]$, the newly joined node randomly gets ID between $[ID_1, ID_2]$.

The overhead of newly joined node for getting ID is $b+l$ RTT measurements and $b+l$ vector distance computation, which is very low.

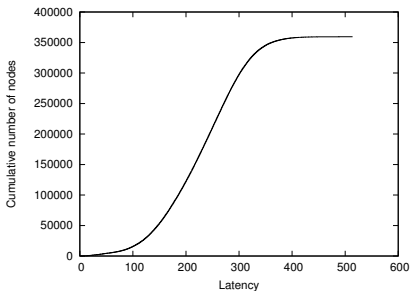

Figure 11: Latency CDF of the transit-stub topology

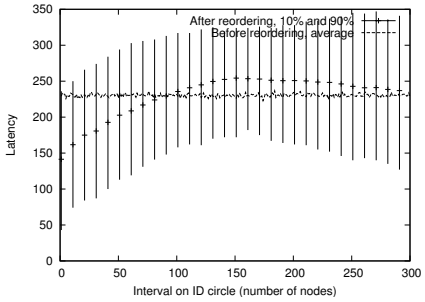

Figure 12: Latency as function of node interval on the ID circle

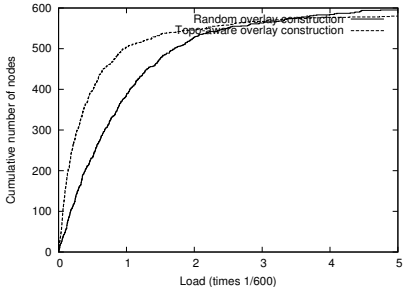

Figure 13: CDF of load (length of ID interval) on each node.

3.3 Experiment results

We ran the overlay construction algorithm on a 600 node transit-stub network. The CDF of latency distribution is shown in figure 11. 50 random nodes in the network are selected as landmarks. The algorithm picks 6 of them as level-1 landmarks and each of the remaining is assigned to one of the 6 as level-2 landmarks. Figure 12 shows the average latency as a function of the number of nodes between the two nodes on the ID circle, using the topology-aware overlay construction algorithm and random construction. It shows that the algorithm effectively assigns near IDs to nearby nodes in the network. The average latency between two adjacent nodes on the ID circle is 141, 61% of the whole-network average 230. Figure 13 shows the CDF of load (length of ID range) of the nodes. The loads are less balanced than the random case, but still pretty good.

Finally, we simulated 10,000 lookups on this network, using finger-only routing, without server-selection. Results are shown in the following table. Stretch is reduced by 22%. This shows that using pretty simple topology-aware overlay construction, we can achieve significant reduction in lookup stretch.

Construction Method	Path Len	Stretch
Random	4.45	2.73
Topology-aware	4.3	2.14

References

[1] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research, 2002.

[2] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP* (2001).

[3] Gonzalez, R. C., and Woods, R. E. *Digital image processing.* 1992.

[4] Hildrum, K., Kubiatowicz, J., Rao, S., and Zhao, B. Distributed data location in a dynamic network. In *Proc. of ACM SPAA* (2002).

[5] MIT Chord implementation. `http://www.pdos.lcs.mit.edu/chord/`.

[6] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. A scalable content-addressable network. In *Proceedings of SIGCOMM* (August 2001), ACM.

[7] Ratnasamy, S., Shenker, S., and Stoica, I. Routing algorithms for DHTs: Some open questions. In *Proc. of IPTPS* (2002).

[8] Rhea, S., Roscoe, T., and Kubiatowicz, J. Structured peer-to-peer overlays need application-driven benchmarks. In *Proc. of IPTPS* (2003).

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

[11] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. Tech. rep., Massachusetts Institute of Technology, 2002.

[12] Zhao, B. Y., Duan, Y., Huang, L., Joseph, A., and Kubiatowicz., J. Brocade: Landmark routing on overlay networks. In *Proc. of IPTPS* (2002).