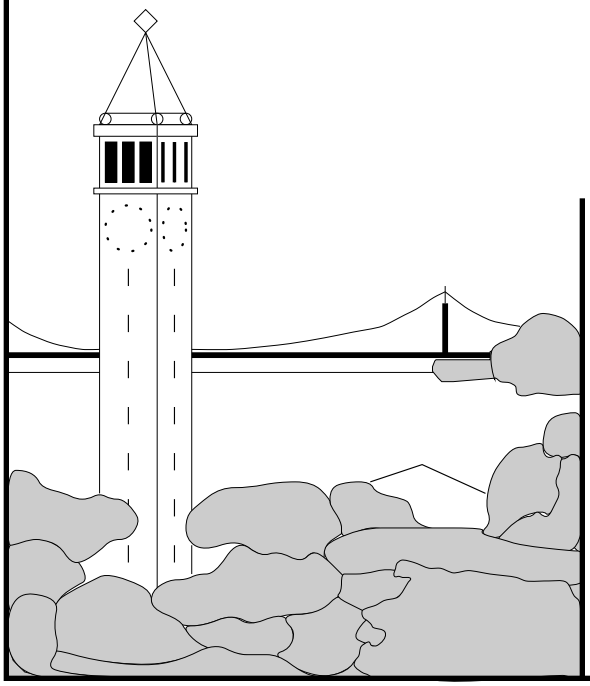
Technical Report: Low Stretch between Nearby Peers*

Kirsten Hildrum, John D. Kubiatowicz, Jeremy Stribling
{hildrum, kubitron}@cs.berkeley.edu, strib@csail.mit.edu



Report No. UCB/CSD-04-1328

June 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

* Supported by NSF career award ANI-9985250, NFS ITR award CCR-0085899, and California MICRO award 00-049, and NSF Cooperative Agreement No. ANI-0225660.

Abstract

A decentralized object location and routing data structure (or DOLR) locates copies of objects in peer-to-peer networks. An efficient DOLR finds nearby copies of objects when possible. The measure of efficiency is stretch, the ratio of the distance traveled to find an object to the distance to the closest copy. Previous empirical work has shown that achieving low stretch is more difficult when objects are nearby, and here we give one reason why this is the case.

Second, one of the important primitives for building a DOLR is finding a nearby peer in the peer-to-peer network. We compare two techniques for finding the nearest neighbor in a peer-to-peer network.

1 Introduction

A peer-to-peer object location system is an evolving set of computers cooperating to store objects. In this technical report, we analyze the *stretch* in such a system. Stretch is the ratio of the distance traveled in the system to find a copy of the object to the direct distance to the closest copy. We describe a simple network with low overall stretch (i.e., the average stretch over all pairs) but high stretch between nearby pairs, and then show via simulation that the simple example is relevant to real networks. Building a low-stretch network requires finding a nearby neighbor as a primitive. In a second section, we show that a system's ability to do this well depends on the underlying network.

Performance in the local area is particularly important when many paths can stay entirely within the local area. This occurs when searching for a popular object in a system that publishes pointers rather than objects (i.e., a DOLR rather than a DHT [DZD+03]) when objects are placed near their points of access. (That is, the Berkeley room-reservation list is placed in the Berkeley subnet.) In the case of some data types (like music), some objects are much more popular than others. For the popular files, there are usually nearby peers sharing the file. Focusing on local area performance is also important when the objects represent services rather than files, since it may be important to find a nearby instance of the desired service.

In particular, large distances between nearby nodes often indicate an object location query path that leaves the local area unnecessarily. Such paths not only present performance problems revealed by stretch measurements, but will also consume limited and expensive wide area bandwidth, and the traffic will be more likely to suffer from router queuing effects and dropped packets. Thus, keeping a query in the local area can greatly increase overall performance.

Two properties have been used to quantify performance in the local area. One such measure is *stretch*, defined above as the ratio of the distance traveled in the system to find a copy of the object to the direct distance to the closest copy. A second property is *local convergence*. A system is said to have local convergence when search paths for a given object tend to meet up quickly. (That is, if A and B are nearby, and they search for the same item, most of the search path should be the same.) Low stretch at every distance implies local convergence, but not vice-versa.

In the following sections, we present an example showing that the effectiveness of techniques for low stretch depend to on the structure of the underlying network.

2 Stretch at Short Distances

In this section, we show in a simple (and unrealistic) example that the overall stretch (i.e., the average stretch over all pairs) is essentially unrelated to stretch between close pairs. Second, we present simulation results showing that this simple example is saying something about the real system.

Place n overlay nodes at the integers on the number line, from 1 to n, such that adjacent nodes are separated by a distance of one. (We assume that n is a power of 2.) A particular object ID and the routes taken to that ID through the overlay create a logical tree on these nodes. Specifically, in a base-2 Tapestry tree, roughly one-half of the nodes are in the bottom level in the tree, one-quarter are in the next level, etc. Suppose the tree is "perfect", meaning that exactly every other node is a leaf, and consider the pair of nodes

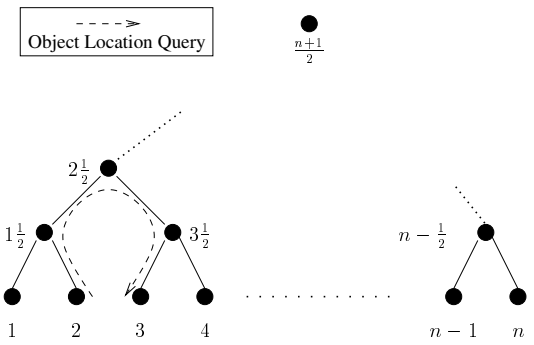


Figure 1: *A perfect base-2 Tapestry tree, with hypothetical average parents..* A sample object location query with an stretch of three is shown.

$(2k-1, 2k)$, for some $k > 0$. If $2k$ is a leaf, then $2k$'s distance to its parent $2k-1$ is one, and $2k-1$'s distance to its parent (itself) is zero; the average distance from a node to its parent is $\frac{1}{2}$. To simplify the calculation, imagine a hypothetical average parent located at a distance of $\frac{1}{2}$ from both nodes, at $2k - \frac{1}{2}$. These are not physical nodes; rather, they represent the average length of the first hop of a route. We repeat this process at each level of the logical routing tree (see Figure 1).

In Tapestry, objects are *published* by placing pointers to them at each node along the path from the publisher to the root, and object location proceeds by checking for pointers along the path from the query source to the root. The stretch is determined by where the publish and search path first intersect, or in other words, the least common ancestor of the publisher and the query source.

We now calculate the average location stretch when the publisher and searcher are at distance one from each other. Half of such pairs of nearby nodes share the same parent, and so have location stretch of one, since the request need only travel a distance of one to reach the publisher. One quarter of the pairs share the same grandparent, but not the same parent; for these nodes, the stretch is three (see Figure 1 for an example). In general, if $i = \log_2 n$, for any positive $j \leq i$ there are 2^{i-j} pairs with location stretch $2^j - 1$. The average stretch is then $\frac{1}{n}\sum_{j=1}^{i} 2^{i-j}(2^j - 1)$, or $i - \sum_{j=1}^{i} \frac{1}{2^j}$, which is less than or equal to $\log n - 1$. Thus, the average stretch is $O(\log n)$. A similar argument shows that for pairs at distance f, the average stretch is $O(\lceil \log(n/f) \rceil)$. Thus, for pairs at distance at least $n/2$, the stretch is constant. Since there are a large number of pairs at distance $n/2$, the effect of these far away pairs effectively drowns out the stretch at shorter distances.

The intuition behind the math is that a given point k needs to be able to route quickly to both $k+1$ and $k-1$, and a binary tree only allows it to share a parent with one or the other, forcing it to pay more to reach the other nearby node. This choice happens at all levels of the tree.

Claim 1 *Realistic topologies that have short stretch at long distances may still long stretch between nearby pairs.*

The argument for the line can be extended to networks that are based on d-dimensional grids. In fact, this problem becomes greater, since in a d-dimensional grid a point is "closer" to more points, and may have to choose among many more different directions than two. (In fact, building structures like this is related to finding HSTs [Bar96, FRT03], for which there is a known lower bound.) To reduce stretch, objects could be published to more than one "close" peer, (by having two parents, for example). However, the number of such additional parents could be quite large–and in the extreme case, may include the whole network. It may be that networks force a certain space-stretch trade off, at least for networks that spread the load equally among participants.[1]

Given the simplicity of the thought experiment above, it may be surprising that the simulation results presented in Figure 2 confirm that this indeed a problem for more realistic topologies. On the left, it shows that the 90th percentile stretch goes down as the distance between the source and destination go up. (The same is true of the mean and median stretch.) The histogram on the right plots the stretch for nearby pairs. (See [SHK03] for more details.)

[1]It may be possible to make the preceding statement formal using techniques similar to those in [KL04].

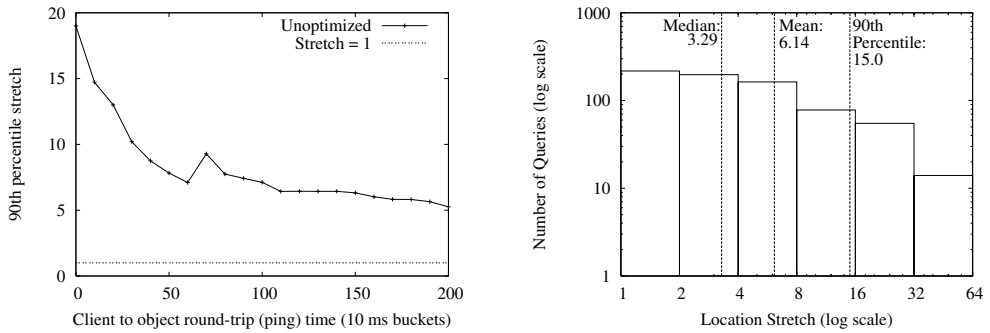


Figure 2: *Tapestry stretch in a simulated transit-stub topology of 1092 nodes.* The left shows the average stretch as a function of the distance between the pairs. The right looks only at pairs within 25 ms and gives the distribution of location stretch in buckets of exponentially-increasing size. The height of each bucket represents the number of queries between such nodes with a stretch of corresponding magnitude.

This histogram shows that although the location stretch of most queries between nearby nodes is small, for some queries it is very large. In fact, this is what our thought experiment predicts: since the stretch experienced by a pair of nodes depends on their least common ancestor in the tree, we would expect the number of pairs with a least common ancestor at level k to decrease exponentially in k, while the stretch experienced should increase exponentially. Note also that there is nearly a factor of two difference between the mean and median (and more than a factor of four between the median and the 90^{th} percentile), suggesting that the local area stretch measurements experienced by lookups varies widely–again a prediction from the simple case. The overall mean stretch for queries between any two nodes is just 3.01 (less than half of the mean stretch for close objects), demonstrating that overall measurements obscure information about the stretch between nearby nodes.

3 PNS(k) and network structure

A peer-to-peer network with Proximity Neighbor Selection [GGG+03] attempts to use nearby neighbors when possible. It is, however, not at all obvious how to find nearby neighbors in a peer-to-peer network. Gummadi et al. [GGG+03] propose a technique called PNS(k). It picks k possible neighbors uniformly at random from the entire network, and then chooses the closest of these k samples to be the chosen neighbor. Gummadi et al. [GGG+03] show that, in practice, this algorithm performs nearly as well as perfect neighbor selection algorithms.

However, they also show that the performance depends on the size of k relative to the the desired "closeness". Gummadi et al. [GGG+03] depend on this implicitly when they simulate a system of 65536 nodes that contains a stub domain of m nodes. Their Figure 11 plots the local convergence of routing paths for some neighbor selection algorithms as m varies. In their simplified model of the network, local convergence is measured by whether or not paths for the same object meet before leaving the stub. But the local convergence of the PNS(16)+PRS scheme depends on m; by looking at the figure, one can see that the turning point is roughly when $m \geq 65536/16 = 4096$. This is exactly what one might expect–when 16 samples are taken, one will likely be within the stub, and two paths for the same object converge within the stub.

In this technical report, we look a little deeper at the performance of PNS(k). We use inter-node latency measurements gathered for 2051 nodes by Dabek and Li using the method from [GSG02].[2] We call this the King data set. Figure 3 shows the performance of PNS(k) on this data set. This suggests that PNS(k) may need many samples to perform well.

We compare this technique to the backward routing technique described by Castro et al. [CDHR02] and Hildrum et al. [HKRZ02]. (The algorithm described by the two sets of authors differ somewhat in the details.) They use the routing structure as follows. Starting with some peer A in the network, consider all

[2] http://www.pdos.lcs.mit.edu/p2psim/kingdata/

	stretch	
k	mean	median
1	85.8	70
2	59.7	56
4	40.3	32.5
8	28.1	23
16	21.2	17
32	14.8	12
64	9.7	6
128	6.0	3

Figure 3: *PNS(k) performance on the King data set.* Notice that k must be quite large get a stretch less than 10. This makes sense, as only 1-5% of the nodes are within distance 10 or 20 of a given peer.

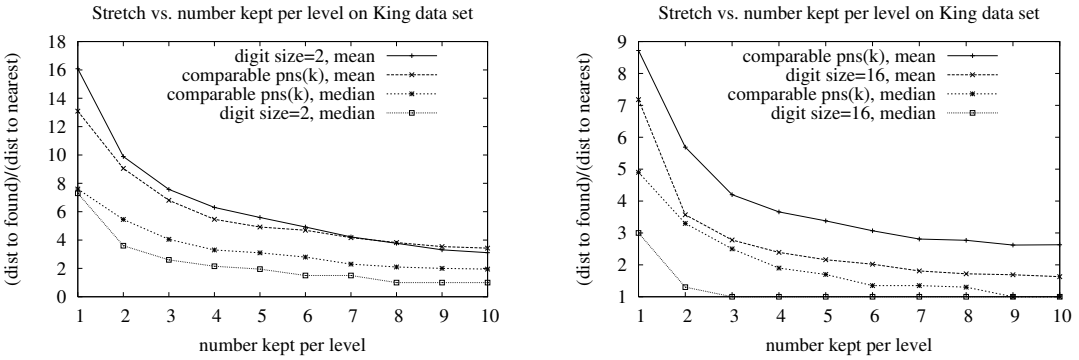


Figure 4: *Comparison of Tapestry with digit sizes of 2 and 16 to PNS(k).* When the digit size is 2, PNS(k) compares favorably with the Tapestry technique. When the digit size is 16, Tapestry is better than PNS(k). The x-axis shows the number of nodes kept per level in the algorithm.

the nodes that use could use A as a next hop. From those, pick the closest l to the querying node. Then, look at all the nodes that could use that as a next hop, and repeat. We call this "backwards-routing" because they use the routing mesh backwards. This algorithm is naturally parameterized in terms of l, the number of nodes kept at each level. Because of the locality-preserving structure of the Pastry and Tapestry trees, this is different than selecting some random peers in the network. There are proofs of correctness for a more complicated version of this algorithm [HKRZ03, HKMR04, HKR03].

We then compare the backwards routing algorithm to PNS(k), as follows. For every test point, we run the Tapestry algorithm, counting the number of distance queries. We then allow PNS(k) to make exactly the same number of queries. The results of this comparison are shown in Figure 4.

When the Tapestry digit size is two, these two algorithms are close in performance. Given the simplicity of PNS(k), this suggests that PNS(k) is the right choice. When the digit size is 16, the difference between the Tapestry search algorithm and PNS(k) is larger, but difference is not great enough to make the Tapestry technique a clear win.

To check the consistency of the results, we ran the same test on a 50,000 node network in Euclidean space. Figure 5 shows that the Tapestry neighbor search algorithm is clearly better in this context. Notice that even when the algorithm keeps only one per level, the median Tapestry stretch is 1, which means at least half the queries return the exactly correct answer! On the other hand, the comparable PNS(k) returns an answer with a much higher stretch. Thus, the network model matters a great deal.

This difference could be due to peculiarities of the King data set. In particular, the King data contains surprisingly large violations of the triangle inequality; it is not clear whether this is the effect of the data collection method or really represents the Internet structure. However, some related works suggests that the King data set may be representative. Rhea et al. [RGRK, RGRK03] examine at the performance of nearest neighbor techniques as part of a DHT in a dynamic network, and in their case, the performance of the

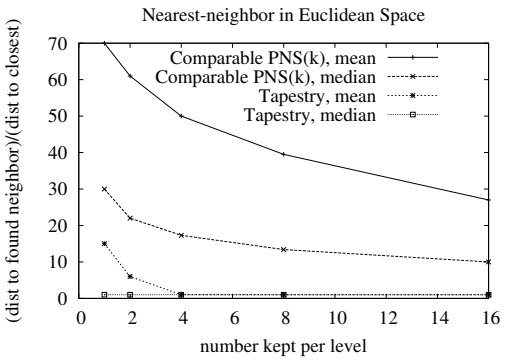


Figure 5: *Tapestry neighbor search vs. PNS(k) in Euclidean Space.*A comparison of the neighbor search algorithm of Tapestry and the comparable PNS(k). The network contains 50,000 nodes, and the search was run from 1000 random points.

Tapestry neighbor algorithm is essentially equivalent, and perhaps slightly worse, than the performance of PNS(k). The fact that they get similar results using a different network model (it uses ModelNet [VYW+02]) suggests that the mediocre performance of the Tapestry neighbor algorithm compared to PNS(k) cannot be blamed on the King data set. However, both the results of [RGRK,GGG+03] and the King results presented here used networks in the thousands,[3] so scale may be partial explanation.

The main disadvantage of the backward-routing techniques is the high number of pings, even when only one node is kept per level. It may be possible to eliminate this problem with the use of synthetic coordinates (as in [CDK+03], for example), which allow approximate distance measurements without network traffic.

While this technique has been analyzed in terms of PRR-like networks, the same backward routing technique may be applicable in other proximity-aware networks (such as a ring with PNS [GGG+03]) to get a structure that could be used to achieve similar performance.

4 Conclusion

This paper provides two examples of ways in which the locality properties of peer-to-peer networks depend on the underlying network configuration. First, we show using a simple example that stretch at short distances may be unreasonably high even when overall stretch is quite low. Though the line example seems to be quite different from a real network, it does predict the results from an experiment on a transit-stub network.

Second, we argue that PNS(k) is probably not the right method to find nearby neighbors in locality-aware networks. We present data that shows a long-standing technique performing better given the same number of distance measurements.

Acknowledgments

Thanks to Satish Rao who pointed out that the worst-case stretch on a tree is high.

References

[Bar96] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 184, 1996.

[3] Rhea et al. [RGRK] uses a network with 10,000 nodes, but it picks only 1000 to participate in the protocol.

[CDHR02] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Anthony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 52–55, June 2002.

[CDK+03] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003. ACM SIGCOMM.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured P2P overlays. In *Proceedings of IPTPS*, pages 33–44, 2003.

[FRT03] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 448–455, 2003.

[GGG+03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. of SIGCOMM*, pages 381–394, 2003.

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, 2002.

[HKMR04] Kirsten Hildrum, John Kubiatowicz, Sean Ma, and Satish Rao. A note on finding the nearest neighbor in growth-restricted metrics. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 553–554, 2004.

[HKR03] Kirsten Hildrum, John Kubiatowicz, and Satish Rao. Another way to find the nearest neighbor in growth-restricted metrics. Technical Report UCB/CSD-03-1267, UC Berkeley, 2003.

[HKRZ02] Kirsten Hildrum, John Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, pages 41–52, 2002.

[HKRZ03] Kirsten Hildrum, John Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. *Theory of Computing Systems*, March 2003.

[KL04] R. Krauthgamer and J. R. Lee. The black-box complexity of nearest neighbor search. In *31st International Colloquium on Automata, Languages and Programming*, July 2004. To appear.

[RGRK] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. To appear in USENIX '04 Annual Technical Conference.

[RGRK03] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB//CSD-03-1299, UC Berkeley, 2003.

[SHK03] Jeremy Stribling, Kirsten Hildrum, and John D. Kubiatowicz. Optimizations for locality-aware structured peer-to-peer overlays. Technical Report UCB/CSD-03-1266, UC Berkeley, Computer Science Division, August 2003.

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, December 2002.