

# Intelligent Neighbor Selection in P2P with CapProbe and Vivaldi

Michael Parker, Amir Nader-Tehrani, Alok Nandan, Giovanni Pau  
Computer Science Department  
University of California Los Angeles  
Los Angeles, CA 90095-1596

**Abstract**—In this paper we propose a new technique for peers on a Pastry network to find optimal entries for their routing tables, where a node is optimal if it minimizes ping time, maximizes bandwidth, or attempts to do both. To assist in such neighbor selection, we investigate using Vivaldi in conjunction with CapProbe. A variety of cost functions, or criteria to judge peers, is devised and evaluated through simulation. We find that intelligent peer selection, leading to optimized routing table entries, can have benefits of halving end-to-end latency and more than doubling end-to-end throughput.

**Index Terms**—Pastry, CapProbe, Vivaldi, intelligent neighbor selection, routing table optimization

## I. INTRODUCTION

Intelligent neighbor selection is an important yet often-overlooked aspect of peer-to-peer networks. *Neighbor selection* refers to a host being able to select one of many peers as a neighbor while still maintaining the correctness of the network topology. *Intelligent* neighbor selection refers to picking the peer that minimizes some associated cost function, where a cost function quantifies the “worth” of a peer according to measurable metrics such as capacity and latency. The driving idea behind intelligent neighbor selection is that, if each node on the network greedily chooses for itself neighbors that minimize a network-wide cost function, then end-to-end measurements for the metrics associated with the cost function will improve.

Most overlay topologies concentrate on minimizing end-to-end latency and maximizing end-to-end capacity by reducing the order of hops traversed when a message is sent. Most popular Distributed Hash Table (DHT) topologies today [4], [8], [13] perform routing in  $O(\log N)$  steps while keeping  $O(\log N)$  connectivity per node. Recently, efforts have been made to reduce these logarithmic bounds, introducing either  $O(1)$  routing [1], [12] or  $O(1)$  state [2], [10]. These proposals, however, introduce new complexity, have either increased costs of joining, leaving, and maintenance, or place more importance on some peers than others to achieve their ends. To improve performance while avoiding such compromises, recent research [6], [11], [15], [16] has focused on the incorporation of *proximity neighbor selection*, or PNS, where a peer chooses from a set of potential neighbors the one with the lowest latency.

Intelligent neighbor selection can be seen as the next logical step beyond PNS, where peers are chosen according to some chosen cost metric that can incorporate latency, bandwidth,

or other quantifiable measures. Customizing the cost metric by which you judge the quality of peers allows one to tailor a peer-to-peer network for a specific application without changing the underlying routing algorithm: Latency-bound applications such as chat programs or networked games could be optimized to minimize round trip time, while throughput-bound applications such as multicast video streaming or worldwide content replication could be optimized to maximize capacity.

This paper makes the following contributions: We demonstrate how, by using the underlying geometry of the Pastry DHT, a node can quickly and efficiently find sets of nodes on which it can perform intelligent neighbor selection. Furthermore, we show how by using the Vivaldi distributed coordinate system and the CapProbe capacity estimation tool, we can devise simple cost functions that optimize routing table entries for end-to-end latency, throughput, or a balance of both. Finally, a global tuning algorithm is presented, which prevents a node on a static network from keeping a local minimum as its optimal routing table entry, forcing it to continue searching for the global minimum elsewhere along the ring.

The rest of the paper is organized as follows: Section II describes the Pastry DHT, and the Vivaldi and CapProbe algorithms we employ to optimize it. We then describe in Section III our algorithm for optimizing the network given a cost function, followed by descriptions of all the cost functions we consider in this paper. Section IV details our simulation environment, including parameters used, and the resulting round trip time and capacity characteristics of a 1740 node network when optimized under each cost function. Finally, Section V describes what we will build upon in future papers, while Section VI concludes this paper.

## II. BACKGROUND

In this section we outline the two key techniques that enable us to select peers intelligently. CapProbe is an accurate, lightweight path capacity estimation tool, while Vivaldi allows us to estimate the round-trip time to a remote node without explicit pinging.

### A. CapProbe

CapProbe [3] is an accurate link capacity estimation tool that uses *Packet Pairs*. Packet Pairs, as the name suggests, is a pair of back-to-back packets that are sent over the any network

path to estimate the path’s characteristics. The basic Packet Pair relies on the fact that if two packets are sent back-to-back and are queued one after another at the narrow link, they will exit the link with a *dispersion*  $T$  given by  $T = \frac{L}{B}$ , where  $L$  is the size of the second packet and  $B$  is the bandwidth of the narrow link, i.e., the capacity-limiting link. If the two packets have the same size, their transmission delays are the same. This means that after the narrow link, a dispersion of  $T$  will be maintained between the packets even if faster links are traversed downstream of the narrow link. This is shown in Figure 1, where  $S$  is the source,  $D$  is the destination, and link  $A-B$  is the narrow link. The narrow link capacity can then be calculated as  $B = \frac{L}{T}$ . The Packet Pair algorithm assumes that the packets will queue next to each other at the narrow link. The presence of cross-traffic can invalidate this assumption.

The underlying idea of CapProbe is that at least one of the two probing packets must have queued if the dispersion at the destination has been distorted from that corresponding to the narrow link capacity. This means that for samples that estimate an incorrect value of capacity, the sum of the delays of the packet pair packets, which we call the *delay sum*, includes cross-traffic induced queuing delay. This delay sum will be larger than the minimum delay sum, which is the delay sum of a sample in which none of the packets suffers cross-traffic induced queuing. The dispersion of such a packet pair sample is not distorted by cross-traffic and will reflect the correct capacity. Based on this observation, CapProbe calculates delay sums of all packet pair samples and uses the dispersion of the sample with the minimum delay sum to estimate the narrow link capacity.

### B. Vivaldi

Vivaldi [17] is an algorithm for assigning synthetic coordinates to hosts such that the distance between the coordinates of two hosts accurately predicts the round trip time, or RTT, between them. Unlike previous methods to allow estimation of latency a priori, Vivaldi is distributed among participating hosts and thus scales efficiently as the size of the network grows. Furthermore, coordinates change quickly in reaction to adverse traffic patterns such as congestion.

To briefly describe how Vivaldi works, each host maintains a set of coordinates and an error estimate denoting the accuracy of these coordinates. The error estimate is derived from how well the host’s coordinates, when combined with coordinates of remote nodes, estimate the RTT to the remote nodes. If, after a sample, the RTT to another node is not equal to the distance between them, the sampling node moves a fraction toward the coordinates that would have perfectly predicted the RTT for that sample. Thus, if the sample RTT is less than the predicted value, the nodes move closer in the coordinate space; if the sample RTT is more than the predicted value, the nodes separate farther. Nodes with high error estimates are assumed to not yet have found their correct position within the coordinate space. As such, after a sample they are allowed to take larger leaps toward the coordinates that would have perfectly predicted the sample, and thus explore the coordinate space more freely in hopes of finding accurate coordinates.

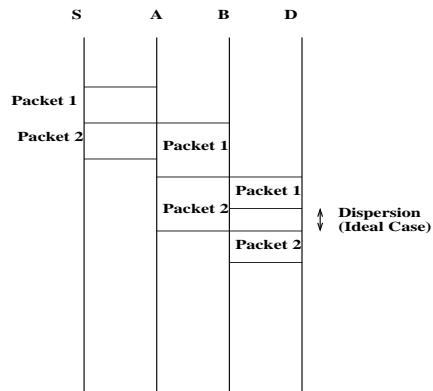


Fig. 1. CapProbe: Ideal Case

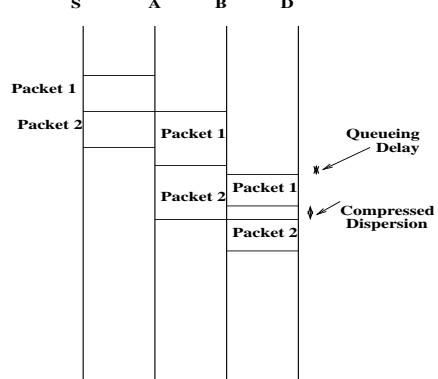


Fig. 2. CapProbe: Over-Estimation of Capacity

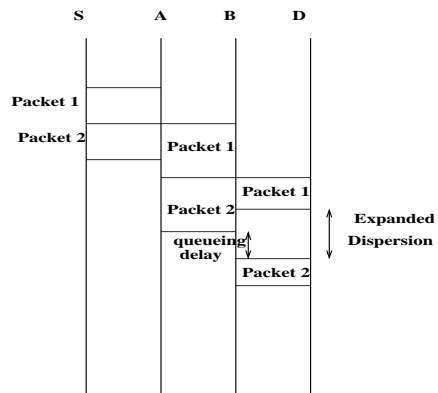


Fig. 3. CapProbe: Under-Estimation of Capacity

### C. Pastry

Pastry [13] is a peer-to-peer overlay topology allowing messages to be routed between  $N$  participating hosts in  $O(\log N)$  time while the connectivity of each host scales with only  $O(\log N)$  complexity. Each host on the network is assigned a randomly chosen Globally Unique Identifier, or *GUID*, in the range  $[0, 2^{160})$ . Such a value is usually derived by the SHA-1 hash its IP address. The service provided by Pastry, like all other Distributed Hash Tables, is to map messages with the same 160-bit key to a single host on the network regardless of the originating node. In Pastry, messages are mapped to the node whose GUID is numerically closest to

the key modulo  $2^{160}$ . The identifier space is thus circular and commonly referred to as a ring. To enable end-to-end delivery, each node on the network must assume responsibility for forwarding messages not meant for it to nodes that are closer to the destination in some sense. In Pastry, the nodes a host knows are found in either its *leaf set* or its *routing table*.

The leaf set consists of the  $\frac{L}{2}$  nodes whose GUIDs immediately precede it in the circular identifier space, and the  $\frac{L}{2}$  nodes whose GUIDs immediately succeed it, where  $L$  is a fixed value shared by all nodes on the network. The routing table, by contrast, logarithmically increases and decreases in size with the size of the network. In our configuration of Pastry, the routing table is organized as 160 rows, indexed from 0 to 159. Treating GUIDs as a sequence of 160 bits, a remote node on the network belongs in row  $i$  of the local host's routing table if its GUID matches the first  $i$  bits of the local node's GUID and differs in the bit at position  $i + 1$ . Such a bit sequence is said to share a *prefix* of length  $i$  with the local node's GUID. Therefore nodes in higher rows of the routing table share successively longer prefixes with the local node's identifier.

The Pastry routing algorithm is as follows: When a Pastry node receives a message, either from a remote host or an application running locally, it first determines whether the message key falls within the range of the identifier space spanned by its leaf set. If this happens, and the local node is the closest node, the message is passed up to the application layer; otherwise, the message is forwarded to the closest remote node in the leaf set, which is necessarily the last hop. If the message key is not spanned by the leaf set, the node finds the length  $i$  of the prefix its GUID shares with the key. If some node exists in row  $i$  of its routing table, it must share a prefix of at least length  $i + 1$  with the message key. This node is thus closer to the key than the current node, and so the message is forwarded on to it. If no node exists in row  $i$ , the message is forwarded to a node whose GUID shares a prefix of length  $i$  with the key and is numerically closer. This way the message is still making progress toward the node that is numerically closest to it.

### III. OPTIMIZATION

#### A. Procedure

As seen in Figure 4, all nodes that begin their identifiers with a given bit sequence  $b$  are found consecutively in the circular identifier space. Therefore, a node whose GUID begins with  $b$  can always find in its leaf set another node whose GUID begins if  $b$  if one exists; we say their leaf sets *overlap*. Now consider an entry in row  $i$  of a given node's routing table; this entry's GUID matches the first  $i$  bits of the given node's GUID, and differs in bit  $i + 1$ . We can conclude that if any other nodes exist with a prefix of length  $i$ , a subset of them must be found in the leaf set of the entry in row  $i$ . Therefore, by *pulling* the leaf set of the entry in row  $i$  of our routing table, we can find a subset of all possible nodes that could go in row  $i$ . We can then choose one that minimizes a selected cost function, and repeat the process for all rows in the routing table.

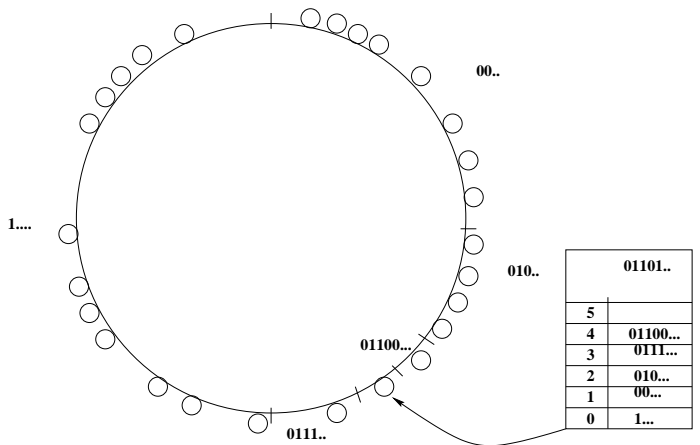


Fig. 4. Pastry DHT

Because GUIDs are assigned to nodes by the SHA-1 hash, although a cluster of nodes may share the same subnet or ISP and therefore have similar IP addresses, their positions in the circular identifier space will be diverse. Thus the nodes in a pulled leaf set should have varied geographic locations, exhibiting a wide array of round trip time and capacity characteristics. This increases our chances of finding a node that has a lower cost with respect to the current node.

Note that simply optimizing each row of our routing table once by the procedure above is not enough, for two reasons: First, if the entry in row  $i$  is not optimal, and a node from its pulled leaf set is chosen to replace it, part of this replacement's leaf set does not overlap with the original entry's leaf set. A node that minimizes the cost function even more than the replacement node may exist there, and so it would be beneficial to optimize again later by pulling the replacement node's leaf set and repeating the process. Second, even if pulling a row entry's leaf set yields no nodes with a lower cost function than that entry, new nodes may later join the network and enter into that entry's leaf set. Alternatively, nodes may later drop from the entry's leaf set, moving entries previously outside its leaf set into it. Either way, we see it would still be beneficial to occasionally pull the leaf set of an optimized entry, hoping to find a new node that has a lower cost function value.

If the *churn rate* of the network is low, however, there is a chance that our algorithm above will stabilize at some entry that is a local minimum. At this point, pulling the leaf set repeatedly of an optimized entry will retrieve the same nodes again and again, all with a higher cost than the current entry. For a network with size  $N$ , however, approximately  $N/2^{i+1}$  could be used to fill row  $i$  of a routing table. Therefore, especially for smaller values of  $i$ , it is likely that some node exists outside the current entry's leaf set that has a lower cost value. Ideally, we would like to find such a node, and even the global minimum. If an entry in row  $i$  of our routing table remains optimal after pulling its leaf set  $k$  times, we turn to a technique inspired by *global tuning* in the Bamboo DHT [7], a variant of Pastry.

In our global tuning phase, we first construct a message with a key sharing a prefix of length  $i$  with our GUID. Like

the current entry in row  $i$ , the key will first differ from our GUID in the bit at position  $i + 1$ . All bits following position  $i + 1$  in the key are randomly generated. Because all bits after position  $i + 1$  are random, there is a high probability that this message will be delivered to a node whose leaf set entries also share a prefix of length  $i$ , but which is also outside the leaf set of our local minimum. The hope is to sample from a set of nodes outside the optimized entry's leaf set. Once delivered by Pastry, the closest node to the message key returns its leaf set. If a node is found in this leaf set with a lower cost value than our current entry in row  $i$ , we choose it as a replacement. Otherwise, row  $i$  remains unchanged and we repeat our global tuning if the current entry is still optimal after pulling its leaf set  $k$  more times.

### B. Cost Functions

Now that we have devised a technique to find suitable routing table entries, we must find some cost function to judge which is best. Here we describe five functions whose effectiveness we will evaluate in the next section:

- **RTT Only:** The node whose Vivaldi coordinates are closest to the current node is chosen.
- **Capacity Only:** CapProbe is run on each node, and the node having the highest capacity is chosen.
- **RTT First:** The  $\frac{L}{2}$  nodes whose Vivaldi coordinates are farthest are discarded. CapProbe is then run on the remaining nodes, and the node having the highest capacity is chosen.
- **Capacity First:** CapProbe is run on each node, and the  $\frac{L}{2}$  nodes whose capacities are lowest are discarded. Of the remaining nodes, the node whose Vivaldi coordinates are closest is chosen.
- **RTT-Capacity Joint Ranking:** CapProbe is first run on each node. After doing this, nodes are ranked separately by their estimated RTT and their bandwidth. Joint rankings are then computed for each node by adding their respective RTT and bandwidth ranks. The node with the highest joint ranking is chosen, while the node with the higher capacity ranking is chosen in the event of a tie.
- **Normalized Deviations:** CapProbe is run on each node to find capacities. The average and standard deviation over all RTTs,  $\overline{rtt}$  and  $\sigma_{rtt}$ , are computed. The average and standard deviation over all capacities,  $\overline{cap}$  and  $\sigma_{cap}$ , are also computed. Peer  $i$  receives a score equal to:

$$s_i = \text{signum}(\overline{rtt} - rtt_i) \left( \frac{\overline{rtt} - rtt_i}{\sigma_{rtt}} \right)^{0.75} + \text{signum}(cap_i - \overline{cap}) \left( \frac{cap_i - \overline{cap}}{\sigma_{cap}} \right)^{0.75}$$

where:

$$\text{signum}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

The score of a node is proportional to how many standard deviations it is from an 'average' node, in terms of both RTT and capacity. Note that we damp both components

by raising each to a power less than unity; this is so nodes with overwhelmingly low latencies or high capacities do not dominate, promoting well balanced entries. The node  $i$  with the highest score  $s_i$  is chosen.

## IV. SIMULATION

### A. Setup

Our simulation starts with a single node on the network. Nodes then join the network one by one, each choosing as its bootstrap node a randomly selected node on the network. Between nodes joining, each node selects up to two nodes it knows of (either from its leaf set or routing table) and pings them, allowing it to refine its own Vivaldi coordinates. The 5D Euclidian coordinate space was used for Vivaldi since it yields the lowest absolute error. A node first optimizes its routing table entries once 30 nodes have joined the network after it, and continues to optimize every time 30 more nodes have joined since its last optimization attempt. The leaf set size  $L$  is 8, while for the purposes of global tuning we set the parameter  $k$  to 3.

This process continues until there are 1740 nodes on the network, at which point we consider the network complete and begin the measurements described below. This leaves some of the last nodes to join the network with inaccurate Vivaldi coordinates and unoptimized routing tables, giving us a snapshot of a network that is still evolving when measurements are performed. Note that such transient conditions are similar to ones encountered in real, deployed peer-to-peer networks exhibiting a high degree of churn and no stabilization period. We therefore feel that the results described below are a good approximation to results from a real networked environment.

At this point, two different nodes are selected at random from the network, and a message is routed between them. We sum the RTTs along the links taken get the end-to-end RTT, and find the link taken with the minimum capacity to get the limiting bottleneck capacity. This process is repeated for 2500 iterations. The average and medians of our measurements are found in Table I and Table II, respectively.

Note that to simulate latency between nodes, we used the King data set [9] provided by MIT, which consists of a full matrix of pair-wise RTTs between 1740 DNS servers collected using the King method [5]. To simulate bandwidth, we created a matrix of randomly generated real numbers having a uniform distribution between 0 and 50. Because the capacity matrix was generated separately from the RTT matrix, we assume no correlation between pair-wise RTTs and capacities. Unlike the RTT matrix, the capacity matrix is asymmetric, modeling links with different upstream and downstream capacities such as cable and DSL.

### B. Results

In Figure 5 we see the end-to-end RTTs between the first 50 source and destination pairs when the network is unoptimized versus when the network is optimized using the RTT Only cost function. We can observe that, on the whole, the end-to-end RTT of a pair is greatly reduced; indeed, RTT Only is especially effective for eliminating outliers in the

unoptimized method (which typically have near a 2000 ms RTT). By Table II, the median RTT is reduced by over 50%, from 741 ms to 331 ms. As expected, the median capacity remains nearly unchanged, as when we generated the pair-wise capacity data we introduced no correlation between it and the RTT data. Similarly, in Figure 6 we see that the end-to-end capacities between the first 50 source and destination pairs increased dramatically when the Capacity Only cost function was used. Again, by Table II, we see that the median end-to-end capacity is increased by over 200%. We note that this percentage increase seems unusually high, and can attribute it to the fact that the capacity distribution was uniformly distributed, when perhaps it follows a Poisson or exponential distribution in real networks.

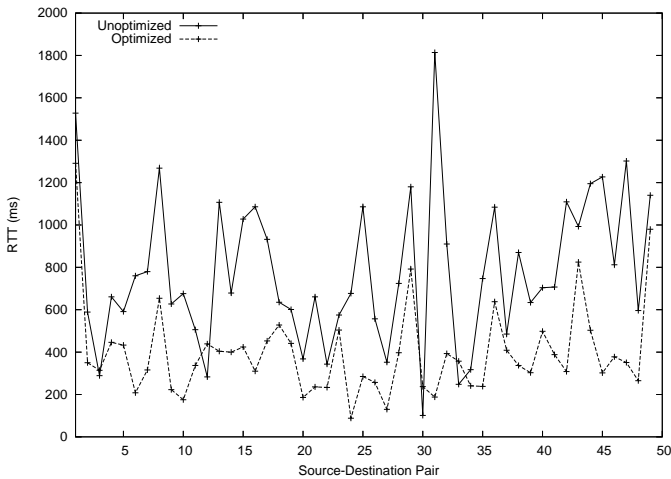


Fig. 5. RTT: Optimized vs. Unoptimized

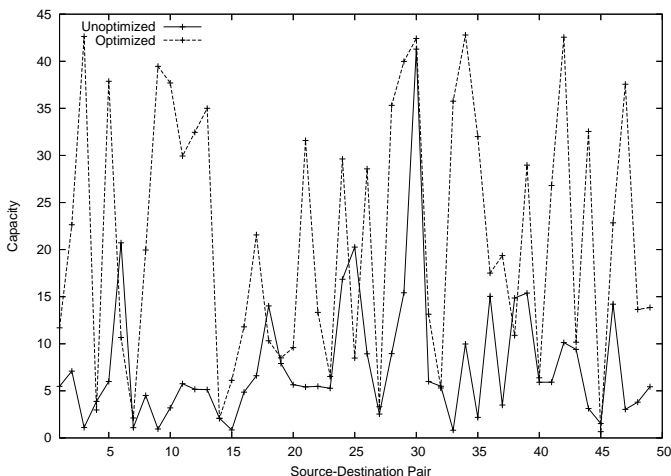


Fig. 6. Capacity: Optimized vs. Unoptimized

One interesting item to note is that, in both graphs, there are pairs that suffer poorer end-to-end performance once optimization is applied! This is due mostly to the greedy approach taken when optimizing nodes are chosen. Let us show an example using the RTT Only cost function: Before optimization, node A routes a message to node B by forwarding the message through intermediate hop  $X_1$ . The end-to-end RTT is thus

equal to  $r_{tt}(A, X_1) + r_{tt}(X_1, B)$ . After optimization, node  $X_2$  replaces  $X_1$  in node A's routing table, and the end-to-end RTT changes to  $r_{tt}(A, X_2) + r_{tt}(X_2, B)$ . There is no guarantee, however, that  $r_{tt}(X_2, B)$  is less than  $r_{tt}(X_1, B)$  because they represent two separate paths through the network. Thus we cannot assert the round-trip-time after optimization is less than the round-trip-time before optimization.

Another reason for this performance degradation is that, as said earlier, some nodes have not yet gone through an optimization round. Consequently, any message forwarded through them goes through an unoptimized node. Still yet other nodes have not found their correct position in the Vivaldi coordinate space. The distance between such a node and another node is thus not a reliable predictor of the RTT between them, which RTT Only relies on for best optimization.

The last four cost functions seek to optimize both end-to-end capacity and latency: RTT First, Capacity First, Joint Ranking, and Normalized Deviations. As we can see again in Table II, each of the four cost functions offers large improvements over an unoptimized network. Note that Joint Ranking and Normalized Deviations maintain a slight advantage over the other two in terms of end-to-end improvement because it is the closest node to the saddle point formed by low RTTs and high capacities in the sample set; RTT First and Capacity First are only approximations to this point. RTT First, however, holds the advantage of using the least amount of overhead; this is because estimating the RTT to a node using Vivaldi coordinates requires no additional traffic and only minor computation, while estimating the capacity to a node requires CapProbe to send packet pairs to it through the network. Since RTT First throws away the  $\frac{L}{2}$  leaf entries with the farthest coordinates before CapProbe is invoked, and the size of a leaf set is  $L$ , we reduce total overhead by approximately 50%.

What's most interesting here is that each method comes

TABLE I  
SIMULATION RESULTS FOR DIFFERENT COST FUNCTIONS

Cost Function	Average	
	RTT	Capacity
Unoptimized	802.07	9.51
RTT Only	379.49	9.35
Capacity Only	861.23	25.41
RTT First	388.15	21.66
Capacity First	410.61	22.51
Joint Ranking	398.98	23.12
Normalized Deviation	403.86	23.85

TABLE II  
SIMULATIONS RESULTS FOR DIFFERENT COST FUNCTIONS

Cost Function	Median	
	RTT	Capacity
Unoptimized	741.00	6.97
RTT Only	331.5	7.27
Capacity Only	786.5	27.03
RTT First	354	22.32
Capacity First	375	23.45
Joint Ranking	367	24.32
Normalized Deviation	371.5	25.19

very close to achieving the end-to-end RTTs and capacities achieved by RTT Only and Capacity Only, respectively. If one wishes to have the end-to-end performance of both RTT Only and Capacity Only without the compromise these three cost functions incur, each node can maintain two routing tables: one whose entries are optimized by RTT Only, and the other whose entries are optimized by Capacity Only. The consequences of this scheme, however, are the need to classify all data as either RTT-link based or Capacity-link based, and the doubling in state and connectivity at each node (although, asymptotically, both remain a tractable  $O(\log N)$ ). We did not include this metric in our paper, as its results are approximately the RTT data from RTT Only and the capacity data from Capacity Only in Table I and Table II.

## V. FUTURE WORK

We hope to improve our work by finding and using real-world capacity data between 1740 hosts in our measurements, as opposed to the uniformly distributed synthetic data used in this paper. If this proves to be too ambitious, another approach we are considering is surveying the capacities of a smaller set of hosts, and using the found distribution in our trials between 1740 hosts. Alternatively, we could run all our experiments on a real network, such as PlanetLab [14]. Some drawbacks to Planetlab, however, include its small size, its geographic concentration in North America, and its abundant low-latency, high-capacity links. These are not necessarily properties of peer-to-peer networks in use today.

We would also like to investigate how varying system-wide parameters affect performance. For example, by intuition, if we increase  $L$  (the size of each node's leaf set), then when a node pulls a routing table entry's leaf set it should see a larger set of nodes with a wider array of round-trip-time and capacity characteristics. It is therefore more likely that the optimization attempt will find a node that has a lower associated cost value than the current entry, and is closer to the global minimum. Other parameters include the global tuning parameter  $k$ , how often nodes pull the leaf sets of their routing table entries, and how often node ping one another to update their Vivaldi coordinates.

Furthermore, we hope to move to an environment that simulates queuing delay, congestion, and churn under periods of optimization. As shown by Rhea et. al [7], ignoring such effects at the stages of simulation can lead to a fragile peer-to-peer network in practice that routes inconsistently, or collapses entirely, when any one condition arises.

## VI. CONCLUSIONS

In this paper, we introduced a new technique for optimizing the entries of a Pastry node's routing table. This procedure was conducted in two steps: First, the current row entry's leaf set was pulled, which we showed had to contain another suitable entry for the given row if one existed. Second, the quality of potential nodes in the leaf set had to be evaluated. We introduced the notion of cost functions to achieve this, where different cost functions emphasized improving different metrics, such as RTT or capacity.

Simulation shows that by each node performing greedy selection of its neighbors using cost functions, end-to-end latency and capacity can be drastically improved. Optimizing strictly for latency, for example, can improve end-to-end performance by over 50%, while optimizing strictly for capacity improves end-to-end performance by over 200%. Surprisingly, attempting to optimize for both metrics does not sacrifice much performance in terms of one or the other, and maintains huge gains over unoptimized networks.

We hope that this paper has brought to light how effective intelligent neighbor selection is in reducing latency and increasing throughput on a peer-to-peer network. Furthermore, we hope it sparks interest in applying the techniques presented here to other DHTs, and devising new cost functions to evaluate the quality of neighboring nodes.

## VII. ACKNOWLEDGEMENTS

We would like to thank Frank Dabek for his infinite patience and insight when answering our questions about Vivaldi, and Matt Rogers for his help in deriving the Normalized Deviation metric.

## REFERENCES

- [1] Achieving One-Hop DHT Lookup and Strong Stabilization by Passing Tokens, B. Leong, J. Li Proceedings of ICON, 2004
- [2] Broose: A Practical Distributed Hash Table Based on the DeBruijn Topology, A.-T. Gai, L. Viennet Proceedings of IEEE P2P, 2004
- [3] CapProbe: A Simple and Accurate Capacity Estimation Technique, R. Kapoor, L.J. Chen, L. Lao, M. Gerla and M.Y. Sanadidi Proceedings of ACM SIGCOMM, 2004
- [4] Chord: A Scalable Peer-to-peer Lookup Service for Distributed Applications, I. Stoica, R. Morris, D. Karger, F. Kaashoek, D. Libel-Nowell, F. Dabek Proceedings of ACM SIGCOMM, 2001
- [5] Estimating Latency Between Arbitrary Internet End Hosts, K. Gummadi, S. Saroiu, S. Gribble Proceedings of SIGCOMM IMW, 2002
- [6] Exploiting Network Proximity in Peer-to-peer Networks, M. Castro, P. Drushel, Y.C. Hu, Antony Rowstron Technical Report MS-TR-2002-82, Microsoft Research, 2002
- [7] Handling Churn in a DHT, S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz USENIX Annual Technical Conference Proceedings, June 2004
- [8] Kademlia: A Peer-to-peer Information System Based on the XOR Metric, P. Maymounkov, D. Mazieres Proceedings of IPTPS, 2002
- [9] "King" Data Set. <http://www.pdos.lcs.mit.edu/p2psim/kingdata>
- [10] Koorde: A Simple Degree-Optimal Hash Table, F. Kaashoek, D. Karger Proceedings of IPTPS, 2003
- [11] Locality Aware Mechanisms for Large-scale Networks, B. Zhao, A. Joseph, J. Kubiatowicz Proceedings of FuDiCo, 2002
- [12] One Hop Lookups for Peer-to-peer Overlays, A. Gupta, B. Liskov, R. Rodrigues Proceedings of HotOS-IX, 2003
- [13] Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, A. Rowstron, P. Druschel IFIP/ACM International Conference on Distributed Systems Platforms, November 2001
- [14] PlanetLab. <http://www.planet-lab.org>
- [15] The Impact of DHT Routing Geometry on Resilience and Proximity, K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoica Proceedings of ACM SIGCOMM, 2003
- [16] Topologically-Aware Overlay Construction and Server Selection, S. Ratnasamy, M. Handley, R. Karp, S. Shenker Proceedings of INFOCOMM, 2002
- [17] Vivaldi: A Decentralized Network Coordinate System, F. Dabek, R. Cox, F. Kaashoek, R. Morris Proceedings of ACM SIGCOMM, 2004