Vivaldi: A Decentralized Network Coordinate System

Frank Dabek, Russ Cox, Frans Kaashoek, Robert Morris
MIT CSAIL
Cambridge, MA
fdabek,rsc,kaashoek,rtm@mit.edu

ABSTRACT

Large-scale Internet applications can benefit from an ability to predict round-trip times to other hosts without having to contact them first. Explicit measurements are often unattractive because the cost of measurement can outweigh the benefits of exploiting proximity information. Vivaldi is a simple, light-weight algorithm that assigns synthetic coordinates to hosts such that the distance between the coordinates of two hosts accurately predicts the communication latency between the hosts.

Vivaldi is fully distributed, requiring no fixed network infrastructure and no distinguished hosts. It is also efficient: a new host can compute good coordinates for itself after collecting latency information from only a few other hosts. Because it requires little communication, Vivaldi can piggy-back on the communication patterns of the application using it and scale to a large number of hosts.

An evaluation of Vivaldi using a simulated network whose latencies are based on measurements among 1740 Internet hosts shows that a 2-dimensional Euclidean model with *height vectors* embeds these hosts with low error (the median relative error in round-trip time prediction is 11 percent).

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Network topology*; C.2.5 [**Computer Communication Networks**]: Local and Wide-Area Networks—*Internet*

General Terms

Algorithms, Measurement, Performance, Design, Experimentation

Keywords

Vivaldi, network coordinates, Internet topology

1. INTRODUCTION

Synthetic coordinate systems [3,17,19,26] allow an Internet host to predict the round-trip latencies to other hosts. Hosts compute synthetic coordinates in some coordinate space such that distance between two hosts' synthetic coordinates predicts the RTT between them in the Internet. Thus, if a host x learns the coordinates of a host y, x doesn't have to perform an explicit measurement to determine the RTT to y; instead, the distance between x and y in the coordinate space is an accurate predictor of the RTT.

The Internet's properties determine whether synthetic coordinates are likely to work well. For example, if Internet latency is dominated by speed-of-light delay over links, and the Internet is well-enough connected that there is a roughly direct physical path between every pair of hosts, and the Internet routing system finds these direct paths, then synthetic coordinates that mimic latitude and longitude are likely to predict latency well.

Unfortunately, these properties are only approximate. Packets often deviate from great-circle routes because few site pairs are directly connected, because different ISPs peer at a limited number of locations, and because transmission time and router electronics delay packets. The resulting distorted latencies make it impossible to choose two-dimensional host coordinates that predict latency perfectly, so a synthetic coordinate system must have a strategy for choosing coordinates that minimize prediction errors. In addition, coordinates need not be limited to two dimensions; Vivaldi is able to eliminate certain errors by augmenting coordinates with a *height*.

The ability to predict RTT without prior communication allows systems to use proximity information for better performance with less measurement overhead than probing. A coordinate system can be used to select which of a number of replicated servers to fetch a data item from; coordinates are particularly helpful when the number of potential servers is large or the amount of data is small. In either case it would not be practical to first probe all the servers to find the closest, since the cost of the probes would outweigh the benefit of an intelligent choice. Content distribution and file-sharing systems such as KaZaA [12], BitTorrent [1], and CoDeeN [31] are examples of systems that offer a large number of replica servers. CFS [6] and DNS [13] are examples of systems that offer modest numbers of replicas, but each piece of data is small. All of these applications could benefit from network coordinates.

Designing a synthetic coordinate system for use in large-scale distributed Internet applications involves the following challenges:

- Finding a metric space that embeds the Internet with little error. A suitable space must cope with the difficulties introduced by Internet routing, transmission time, and queuing.
- Scaling to a large number of hosts. Synthetic coordinate systems are of most value in large-scale applications; if only few hosts are involved, direct measurement of RTT is practical.
- Decentralizing the implementation. Many emerging applications, such as peer-to-peer applications, are distributed and

SIGCOMM'04, Aug. 30–Sept. 3, 2004, Portland, Oregon, USA.

symmetric in nature and do not inherently have special, reliable hosts that are candidates for landmarks.

- Minimizing probe traffic. An ideal synthetic coordinate system would not introduce any additional network traffic, but would be able to gather whatever information it needed from the application's existing communication.

- Adapting to changing network conditions. The relative location of a host in a network may change due to congestion or even reconfiguration of the network. The system should be able to adjust the coordinates of hosts periodically to respond to these changes.

A number of existing synthetic coordinate systems address some of these challenges, but none addresses them all, as Section 6 discusses.

The primary contribution of this paper is a decentralized, low-overhead, adaptive synthetic coordinate system, Vivaldi, that computes coordinates which predict Internet latencies with low error. Vivaldi was developed for and is used by the Chord peer-to-peer lookup system which uses coordinates to avoid contacting distant hosts when performing a lookup [7].

This paper extends earlier descriptions of Vivaldi [4, 5] and considers new variations, particularly a coordinate space that includes the notion of a directionless *height* that improves the prediction accuracy of the system on data sets derived from measurements of the Internet. Height is included to capture transmission delays on the access links of single-homed hosts. A detailed evaluation using a simulator driven with actual Internet latencies between 1740 Internet hosts shows that Vivaldi achieves errors as low as GNP [17], a landmark-based coordinate system, even though Vivaldi has no notion of landmarks.

A further contribution of this paper is that coordinates drawn from a two-dimensional Euclidean model with a height can accurately predict latency between the 1740 Internet hosts. Simulations show that this model is better than 2- or 3-dimensional Euclidean models or a spherical model. These findings suggest that the following properties hold in the data set: inter-host RTT is dominated by geographic distance, the Internet core does not "wrap around" the Earth to any significant extent, and the time required to traverse an access-link is often a significant fraction of total RTT.

The rest of this paper is organized as follows. Section 2 presents the Vivaldi algorithm. Section 3 describes the methodology for evaluating Vivaldi. Section 4 presents the results from evaluating Vivaldi. Section 5 investigates different models to embed the Internet. Section 6 discusses the related work that led us to Vivaldi. Finally, section 7 summarizes our conclusions.

2. ALGORITHM

Vivaldi assigns each host synthetic coordinates in a coordinate space, attempting to assign coordinates such that the distance in the coordinate space between two hosts accurately predicts the packet transmission RTT between the hosts. No low-dimensional coordinate space would allow Vivaldi to predict RTTs between Internet hosts *exactly*, because, for example, Internet latencies violate the triangle inequality. The algorithm instead attempts to find coordinates that minimize the error of predictions.

We first describe this prediction error in more detail and briefly discuss possible coordinate systems. Then, we show a simple centralized algorithm that finds coordinates that minimize a squared error function given complete knowledge of RTTs in the network. Then we present a simple distributed algorithm that computes coordinates based on measurements from each node to a few other nodes. Finally, we refine this distributed algorithm to converge quickly to accurate coordinates.

2.1 Prediction error

Let L_{ij} be the actual RTT between nodes i and j, and x_i be the coordinates assigned to node i. We can characterize the errors in the coordinates using a squared-error function:

$$E = \sum_i \sum_j \left(L_{ij} - \left\|x_i - x_j\right\|\right)^2 \quad (1)$$

where $\left\|x_i - x_j\right\|$ is the distance between the coordinates of nodes i and j in the chosen coordinate space. Other systems choose to minimize a different quantity; PIC [3], for instance, minimizes squared relative error. We chose the squared error function because it has an analogue to the displacement in a physical mass-spring system: minimizing the energy in a spring network is equivalent to minimizing the squared-error function.

2.2 Synthetic coordinate structure

Algorithms can choose the structure of coordinates and the distance function that determines the predicted latency given two coordinates. Coordinates should be compact and it should be easy to compute an RTT prediction given two coordinates. The simplest choice is to use n-dimensional coordinates with the standard Euclidean distance function. Spherical, toroidal, hyperbolic and other coordinate structures have also been proposed (e.g., [27]). These coordinate systems use alternative distance functions in the hope that they predict latency better. Section 5 will present the height-vector coordinates that we propose. In the remainder of this section, however, we will present algorithms that work with any coordinate system that supports the magnitude, addition, and subtraction operations.

2.3 Centralized algorithm

We first describe a simple, centralized algorithm than can minimize Equation 1. Vivaldi is a distributed version of this algorithm. Given our choice of E, simulating of a network of physical springs produces coordinates that minimize E.

Conceptually, this minimization places a spring between each pair of nodes (i, j) with a rest length set to the known RTT (L_{ij}). The current length of the spring is considered to be the distance between the nodes in the coordinate space. The potential energy of such a spring is proportional to the square of the displacement from its rest length: the sum of these energies over all springs is exactly the error function we want to minimize.

Since the squared-error function is equivalent to spring energy, we can minimize it by simulating the movements of nodes under the spring forces. While the minimum energy configuration of the spring system corresponds to the minimum error coordinate assignment, it is not guaranteed that the simulation will find this global minimum: the system may come to rest in a local minimum.

This approach to minimization mirrors work on model reconstruction [11] and a similar recent coordinate approach using force fields [26].

We will now describe the centralized algorithm more precisely. Define F_{ij} to be the force vector that the spring between nodes i and j exerts on node i. From Hooke's law we can show that F is:

$$F_{ij} = \left(L_{ij} - \left\|x_i - x_j\right\|\right) \times \mathrm{u}(x_i - x_j).$$

The scalar quantity $\left(L_{ij} - \left\|x_i - x_j\right\|\right)$ is the displacement of the spring from rest. This quantity gives the magnitude of the force ex-

```
// Input: latency matrix and initial coordinates
// Output: more accurate coordinates in x
compute_coordinates(L, x)
 while (error (L, x) > tolerance)
    foreach i
      F = 0
      foreach j
        // Compute error/force of this spring. (1)
        e = L_ij − ||x_i − x_j||
        // Add the force vector of this spring to the total force. (2)
        F = F + e × u(x_i − x_j)
      // Move a small step in the direction of the force. (3)
      x_i = x_i + t × F
```

Figure 1: The centralized algorithm.

erted by the spring on i and j (we will ignore the spring constant). The unit vector $u(x_i - x_j)$ gives the direction of the force on i. Scaling this vector by the force magnitude calculated above gives the force vector that the spring exerts on node i.

The net force on i (F_i) is the sum of the forces from other nodes:

$$F_i = \sum_{j \neq i} F_{ij}.$$

To simulate the spring network's evolution the algorithm considers small intervals of time. At each interval, the algorithm moves each node (x_i) a small distance in the coordinate space in the direction of F_i and then recomputes all the forces. The coordinates at the end of a time interval are:

$$x_i = x_i + F_i \times t,$$

where t is the length of the time interval. The size of t determines how far a node moves at each time interval. Finding an appropriate t is important in the design of Vivaldi.

Figure 1 presents the pseudocode for the centralized algorithm. For each node i in the system, compute_coordinates computes the force on each spring connected to i (line 1) and adds that force to the total force on i (line 2). After all of the forces have been added together, i moves a small distance in the direction of the force (line 3). This process is repeated until the system converges to coordinates that predict error well.

This centralized algorithm (and the algorithms that will build on it) finds coordinates that minimize squared error because the force function we chose (Hooke's law) defines a force that is proportional to displacement. If we chose a different force function, a different error function would be minimized. For instance, if spring force were a constant regardless of displacement, this algorithm would minimize the sum of (unsquared) errors.

2.4 The simple Vivaldi algorithm

The centralized algorithm described in Section 2.3 computes coordinates for all nodes given all RTTs. Here we extend the algorithm so that each node computes and continuously adjusts its coordinates based only on measured RTTs from the node to a handful of other nodes and the current coordinates of those nodes.

Each node participating in Vivaldi simulates its own movement in the spring system. Each node maintains its own current coordinates, starting with coordinates at the origin. Whenever a node communicates with another node, it measures the RTT to that node and also learns that node's current coordinates.

The input to the distributed Vivaldi algorithm is a sequence of

```
// Node i has measured node j to be rtt ms away,
// and node j says it has coordinates x_j.
simple_vivaldi(rtt, x_j)
 // Compute error of this sample. (1)
 e = rtt − ||x_i − x_j||
 // Find the direction of the force the error is causing. (2)
 dir = u(x_i − x_j)
 // The force vector is proportional to the error (3)
 f = dir × e
 // Move a a small step in the direction of the force. (4)
 x_i = x_i + δ × dir
```

Figure 2: The simple Vivaldi algorithm, with a constant timestep δ.

such samples. In response to a sample, a node allows itself to be pushed for a short time step by the corresponding spring; each of these movements reduce the node's error with respect to one other node in the system. As nodes continually communicate with other nodes, they converge to coordinates that predict RTT well.

When node i with coordinates x_i learns about node j with coordinates x_j and measured RTT rtt, it updates its coordinates using the update rule:

$$x_i = x_i + \delta \times \Big(rtt - \|x_i - x_j\|\Big) \times u(x_i - x_j).$$

This rule is identical to the individual forces calculated in the inner loop of the centralized algorithm.

Because all nodes start at the same location, Vivaldi must separate them somehow. Vivaldi does this by defining u(0) to be a unit-length vector in a randomly chosen direction. Two nodes occupying the same location will have a spring pushing them away from each other in some arbitrary direction.

Figure 2 shows the pseudocode for this distributed algorithm. The simple_vivaldi procedure is called whenever a new RTT measurement is available. simple_vivaldi is passed an RTT measurement to the remote node and the remote node's coordinates. The procedure first calculates the error in its current prediction to the target node (line 1). The node will move towards or away from the target node based on the magnitude of this error; lines 2 and 3 find the direction (the force vector created by the algorithm's imagined spring) the node should move. Finally, the node moves a fraction of the distance to the target node in line 4, using a constant timestep (δ).

This algorithm effectively implements a weighted moving average that is biased toward more recent samples. To avoid this bias, each node could maintain a list of every sample it has ever received, but since all nodes in the system are constantly updating their coordinates, old samples eventually become outdated. Further, maintaining such a list would not scale to systems with large numbers of nodes.

2.5 An adaptive timestep

The main difficulty in implementing Vivaldi is ensuring that it converges to coordinates that predict RTT well. The rate of convergence is governed by the δ timestep: large δ values cause Vivaldi to adjust coordinates in large steps. However, if all Vivaldi nodes use large δ values, the result is typically oscillation and failure to converge to useful coordinates. Intuitively, a large δ causes nodes to jump back and forth across low energy valleys that a smaller delta would explore.

An additional challenge is handling nodes that have a high error

in their coordinates. If a node n communicates with some node that has coordinates that predict RTTs badly, any update that n makes based on those coordinates is likely to increase prediction error rather than decrease it.

We would like to obtain both fast convergence and avoidance of oscillation. Vivaldi does this by varying δ depending on how certain the node is about its coordinates (we will discuss how a node maintains an estimate of the accuracy of its coordinates in Section 2.6). When a node is still learning its rough place in the network (as happens, for example, when the node first joins), larger values of δ will help it move quickly to an approximately correct position. Once there, smaller values of δ will help it refine its position.

A simple adaptive δ might use a constant fraction ($c_c < 1$) of the node's estimated error:

$$\delta = c_c \times \text{local error}$$

δ can be viewed as the fraction of the way the node is allowed to move toward the perfect position for the current sample. If a node predicts its error to be within ±5%, then it won't move more than 5% toward a corrected position. On the other hand, if its error is large (say, ±100%), then it will eagerly move all the way to the corrected position.

A problem with setting δ to the prediction error is that it doesn't take into account the accuracy of the remote node's coordinates. If the remote node has an accuracy of ±50%, then it should be given less credence than a remote node with an accuracy of ±5%. Vivaldi implements this timestep:

$$\delta = c_c \times \frac{\text{local error}}{\text{local error} + \text{remote error}} \quad (2)$$

Using this δ, an accurate node sampling an inaccurate node will not move much, an inaccurate node sampling an accurate node will move a lot, and two nodes of similar accuracy will split the difference.

Computing the timestep in this way provides the properties we desire: quick convergence, low oscillation, and resilience against high-error nodes.

2.6 Estimating accuracy

The adaptive timestep described above requires that nodes have a running estimate of how accurate their coordinates are. Each node compares each new measured RTT sample with the predicted RTT, and maintains a moving average of recent relative errors (absolute error divided by actual latency). As in the computation of δ, the weight of each sample is determined by the ratio between the predicted relative error of the local node and of the node being sampled. In our experiments, the estimate is always within a small constant factor of the actual error. Finding more accurate and more elegant error predictors is future work, but this rough prediction has been sufficient to support the parts of the algorithm (such as the timestep) that depend on it.

2.7 The Vivaldi algorithm

Figure 3 shows pseudocode for Vivaldi. The vivaldi procedure computes the weight of a sample based on local and remote error (line 1). The algorithm must also track the local relative error. It does this using a weighted moving average (lines 2 and 3). The remainder of the Vivaldi algorithm is identical to the simple version.

Vivaldi is fully distributed: an identical vivaldi procedure runs on every node. It is also efficient: each sample provides information that allows a node to update its coordinates. Because Vivaldi is constantly updating coordinates, it is reactive; if the underlying topology changes, nodes naturally update their coordinates accordingly. Finally, it handles high-error nodes. The next sections evaluate how well Vivaldi achieves these properties experimentally and investigate what coordinate space best fits the Internet.

```
// Incorporate new information: node j has been
// measured to be rtt ms away, has coordinates x_j,
// and an error estimate of e_j.
//
// Our own coordinates and error estimate are x_i and e_i.
//
// The constants c_e and c_c are tuning parameters.
vivaldi(rtt, x_j, e_j)
  // Sample weight balances local and remote error. (1)
  w = e_i/(e_i + e_j)

  // Compute relative error of this sample. (2)
  e_s = |||x_i - x_j|| - rtt|/rtt

  // Update weighted moving average of local error. (3)
  e_i = e_s × c_e × w + e_i × (1 - c_e × w)

  // Update local coordinates. (4)
  δ = c_c × w
  x_i = x_i + δ × (rtt - ||x_i - x_j||) × u(x_i - x_j)
```

Figure 3: The Vivaldi algorithm, with an adaptive timestep.

3. EVALUATION METHODOLOGY

The experiments are conducted using a packet-level network simulator running with RTT data collected from the Internet. This section presents the details of the framework used for the experiments.

3.1 Latency data

The Vivaldi simulations are driven by a matrix of inter-host Internet RTTs; Vivaldi computes coordinates using a subset of the RTTs, and the full matrix is needed to evaluate the quality of predictions made by those coordinates.

We use two different data sets derived from measurements of a real network. The first and smaller data set involves 192 hosts on the PlanetLab network test bed [20]. These measurements were taken from a public PlanetLab all-pairs-pings trace [28].

The second data set involves 1740 Internet DNS servers. We built a tool based on the King method [10] to collect the full matrix of RTTs. To determine the distance between DNS server A and server B, we first measure the round trip time to server A and then ask server A to recursively resolve a domain served by B. The difference in times between the two operations yields an estimate of the round trip time between A and B (see Figure 4). Each query involves a unique target name to suppress DNS caching.

We harvested the addresses of recursive DNS servers by extracting the NS records for IP addresses of hosts participating in a Gnutella network. If a domain is served by multiple, geographically diverse name servers, queries targeted at domain D (and intended for name server B) could be forwarded to a different name server, C, which also serves D. To avoid this error, we filtered the list of target domains and name servers to include only those domains where all authoritative name servers are on the same subnet (i.e. the IP addresses of the name servers are identical except for the low octet). We also verified that the target nameservers were responsible for the associated names by performing a non-recursive

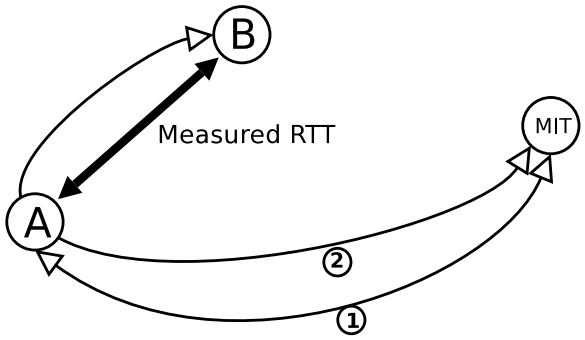

Figure 4: It is possible to measure the distance between two nameservers by timing two DNS queries. The first query (1) is for a name in the domain of nameserver A. This returns the latency to the first nameserver. The second query is for a name in the domain nameserver B (2) but is sent initially to the recursive nameserver A. The difference between the latency of (1) and (2) is the latency between nameserver A and B.

query for that name and checking for the "aa" bit in the response header, which indicates an authoritative answer.

We measured pairwise RTTs continuously, at random intervals, over the course of a week. Around 100 million measurements were made in total. We compute the final RTT for a given pair as the median of all trials. Using the median RTT filters out the effects of transient congestion and packet loss. Other measurement studies [17] have used the minimum measured RTT to eliminate congestion effects; this approach is not appropriate for the King technique since congestion can cause measured RTT to be higher or lower than the true value. King can report a RTT lower than the true value if there is congestion on the path to the first nameserver.

Some nameservers were obvious outliers in the data set: the latency to these servers was equal and small from all hosts. This inaccuracy could be the result of high load on the nameservers themselves or heavy queuing near the servers. If load or queuing at name server A adds a delay that is significantly larger than the network latency, the initial query (to *A*) and recursive query (via *A* to *B*) will require roughly the same amount of time and the estimated latency between that server and any other server will be near zero.

We identified these servers by the disproportionate number of triangle inequality violations they participated in. These servers were removed from the data set. About 10 percent of the original nodes were removed in this way.

The PlanetLab nodes span the globe, but most are located at North American universities with fast Internet2 connections. The King nodes are all name servers, and thus still likely to be well connected to the Internet, but the servers are more geographically diverse than the PlanetLab nodes. The median RTT of the PlanetLab data set is 76ms and of the King data set is 159ms.

We also used two synthetic data sets. The grid data set is constructed to provide a perfect two-dimensional fit; this data set is created by assigning two-dimensional coordinates to each node and using the Euclidean distances between nodes to generate the matrix. When fitting this data set, Vivaldi recovers the coordinates up to rotation and translation.

We also use the ITM topology generation tool [2] to generate topologies. The latency between two nodes in this data set is found by finding the shortest path through the weighted graph that ITM generates. This data set allows us to explore how topology changes affect Vivaldi.

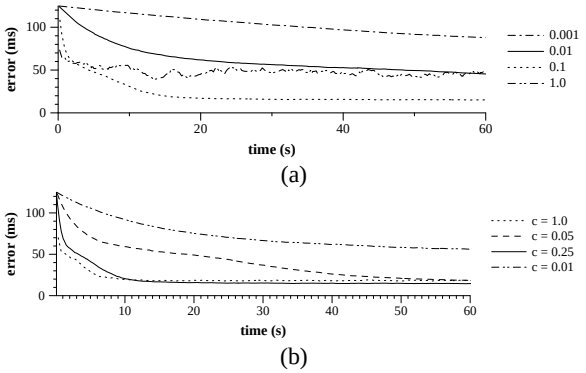

Figure 5: The effect of δ on rate of convergence. In (a), δ is set to one of a range of constants. In (b), δ is calculated with Equation 2, with c_c values ranging from 0.01 to 1.0. The adaptive δ causes errors to decrease faster.

3.2 Using the data

We used the RTT matrices as inputs to a packet-level network simulator [9]. The simulator delays each packet transmission by half the time specified in the RTT matrix. Each node runs an instance of Vivaldi which sends RPCs to other nodes, measures the RTTs, and uses those RTTs to run the decentralized Vivaldi algorithm.

We define the error of a link as the absolute difference between the predicted RTT for the link (using the coordinates for the two nodes at the ends of the link) and the actual RTT. We define the error of a node as the median of the link errors for links involving that node. We define the error of the system as the median of the node errors for all nodes in the system.

The main limitation of the simulator is that the RTTs do not vary over time: the simulator does not model queuing delay or changes in routing. Doing this typically requires modeling the underlying structure of the network. Since this research involves evaluating models for the structure of the network, it seems safest to stick to real, if unchanging, data rather than model a model.

In all of the experiments using the simulator, each node measures a RTT to some other node once each second.

4. EVALUATION

This section examines (1) the effectiveness of the adaptive time-step δ; (2) how well Vivaldi handles high-error nodes; (3) Vivaldi's sensitivity to communication patterns, in order to characterize the types of network applications that can use Vivaldi without additional probe traffic; (4) Vivaldi's responsiveness to network changes; and (5) Vivaldi's accuracy compared to that of GNP. The experiments presented in this section use Euclidean coordinates; Section 5 investigates other coordinate systems.

4.1 Time-Step choice

Section 2.5 claimed that too large a time-step could result in oscillating coordinates with poor ability to predict latency, and that small time-steps would result in slow convergence time. To evaluate this intuition we simulated Vivaldi on the King data set using 3-dimensional Euclidean coordinates.

Figure 5(a) plots the progress of the simulation using various constant values of δ. The plot shows the median prediction error as a function of time. Small values of δ, such as 0.001, cause slow convergence; increasing δ to 0.01 causes faster convergence; but increasing δ again to 1.0 prevents Vivaldi from finding low-error

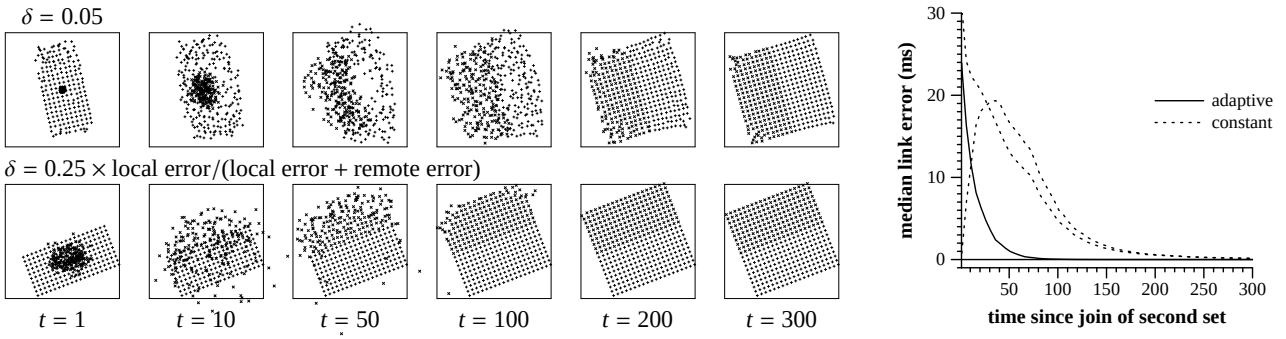

Figure 6: The evolution of a stable 200-node network after 200 new nodes join. When using a constant δ, the new nodes confuse the old nodes, which scatter until the system as a whole has re-converged. In contrast, the adaptive δ (Equation 2) allows new nodes to find their places quickly without disturbing the established order. The graph plots link errors for constant (dotted) and adaptive (solid) δ. At $t = 1$, the lower line in each pair is the median error among the initial nodes. The higher line in each pair is the median error among all pairs. The constant δ system converges more slowly than the adaptive system, disrupting the old nodes significantly in the process.

coordinates. The reason for the high average error is that the high δ causes the coordinates to oscillate in large steps around the best values.

In Figure 5(b) we repeat the experiment using δ as computed in Equation 2. The data show the effectiveness of using a large δ when a node's error is high (to converge quickly) and a small δ when a node's error is low (to minimize the node's oscillation around good coordinates). Empirically, a c_c value of 0.25 yields both quick error reduction and low oscillation.

4.2 Robustness against high-error nodes

Ideally, Vivaldi would cope well with large numbers of newly-joined nodes with inconsistent coordinates. Vivaldi's adaptive δ should address this problem: when a node joins, it knows its relative error is quite large, and so when it communicates with other nodes, those other nodes will approach it with appropriate skepticism.

Figure 6 shows the results of a simulation to test this hypothesis. The simulation uses the two-dimensional grid data set to make it easy to visualize the evolution of the system. The simulation started with 200 nodes that already knew coordinates that predicted latency well. Then we added 200 new nodes to the system and let the system evolve, using $\delta = 0.05$ in one case and Equation 2 with $c_c = 0.25$ in the other. Figure 6 shows the evolution of the two systems as well as the error over time. After a few iterations using the constant δ metric, the initial structure of the system has been destroyed, a result of wise old nodes placing too much faith in young high-error nodes. Because the initial structure is destroyed, existing nodes can no longer use the current coordinates of other existing nodes to predict latency until the system re-converges.

In contrast, the adaptive δ preserves the established order, helping the new nodes find their places faster. Also, because the structure of the original nodes is preserved while new nodes join, those nodes can continue to use current coordinates to make accurate predictions to other original nodes. Finally, the convergence time of the new nodes is significantly faster; they converge at $t = 60$ using the relative time-step versus $t \approx 250$ using the constant δ.

4.3 Communication patterns

As presented, Vivaldi relies on samples obtained from traffic generated by the application using it. To understand the range of systems in which this approach is appropriate, we must characterize the sampling necessary for accurate computation of coordinates.

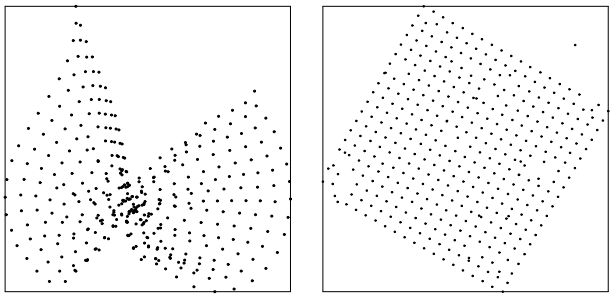

Figure 7: A pathological case showing the possible effect of communication patterns on the chosen coordinates. In the first case, nodes only contact their four nearest neighbors, allowing the resulting coordinates to twist over long distances. In the second case, nodes contact distant nodes as well, improving the accuracy of the coordinates at the larger scale.

Some kinds of sampling work badly. For example, Priyantha *et al.* [21] show that sampling only nearby (low-latency) nodes can lead to coordinates that preserve local relationships but are far from correct at a global scale. Figure 7 shows the coordinates chosen for nodes laid out in a grid when each node communicates only with its four neighbors. This case is clearly a pathological one, but we would like to characterize the boundary between normal behavior and pathology.

The pathological case can be fixed by adding long-distance communications, giving the nodes a more global sense of their place in the network. But how much long-distance communication is necessary in order to keep the coordinates from distorting? To answer this question, we ran an experiment with a grid of 400 nodes. Each node was assigned eight neighbors: the four immediately adjacent to it and four chosen at random (on average, the random neighbors will be far away). At each step, each node decides to communicate either with an adjacent neighbor or a faraway neighbor. Specifically, each node chooses the faraway neighbor set with probability p. Then a specific node to sample is chosen from the set at random.

Figure 8 shows the effect of p on the final accuracy of the coordinates. When half of the communication is to distant nodes, coordinates converge quickly. Convergence slows as the proportion of distant nodes increases, but similarly accurate coordinates are

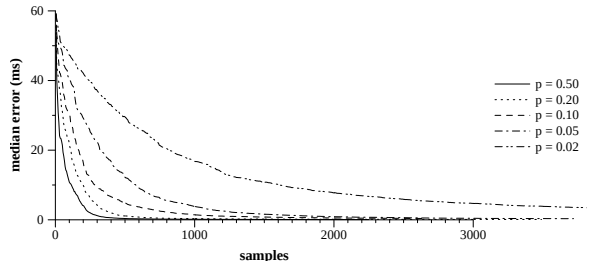

Figure 8: The effect of long-distance communication on the accuracy of the coordinates. Each line in the graph plots prediction error over time for an experiment in which nodes contact distant nodes (as opposed to nearby nodes) with probability p at each time step.

eventually chosen for small proportions of distant nodes, suggesting that even when only 5% of the samples involve distant nodes, skewed coordinate placements like in Figure 7 will be avoided.

4.4 Adapting to network changes

Because Vivaldi constantly recomputes coordinates, it naturally adapts to changes in the network. To test how well Vivaldi responds to changes we created a synthetic "Transit-Stub" topology of 100 hosts using the ITM tool [2]. We used Vivaldi to find coordinates for the 100 hosts; Vivaldi found a 6-dimensional fit using 32 randomly chosen neighbors. We then changed the network topology by increasing the length of one of the stub's connection to the core by a factor of 10. Figure 9 shows the median of the absolute error predictions made by each node over time. Prior to time 100 seconds the nodes have stabilized and the median prediction error is around 15ms. At time 100 the topology is changed to include the much longer transit-stub link. Shortly after the change the median error rises to 25ms the network quickly re-converges (by time 120ms) to a new stable configuration. The error is higher during this time period than at 99ms because the new configuration is more difficult to model. To show this, we restore the original configuration at time 300s. The nodes quickly reconverge to positions which give a median error of again, around 15ms.

Vivaldi also scales to large networks. The algorithm uses a constant amount of storage on each node, and as shown in the previous section can use measurements of traffic that would have been sent anyway. The algorithm has no requirements that scale even logarithmically in the number of nodes, much less linearly.

The one exception is startup cost for an initial network. A network of millions of nodes cannot coordinate itself from scratch in a constant amount of time. However, networks of millions of nodes tend not to spring up all at once. They start as smaller networks and grow incrementally. Once there is a critical mass of well-placed nodes in a Vivaldi network, a new node joining the system needs to make few measurements in order to find a good place for itself. That is, once there are enough nodes in the system, the joining cost for a new node is only a small constant number of network samples, regardless of the size of the network.

To demonstrate this claim, we initialized a 1,000-node network using the King data set. Once that network had converged on accurate coordinates, we added 1,000 new nodes, one at a time, measuring the actual (not estimated) prediction error of each newly placed node as a function of the number of samples obtained. Each new node's prediction error is as low as it will ever be after about 20 samples. New nodes are able to converge quickly because they begin with a large initial time-step.

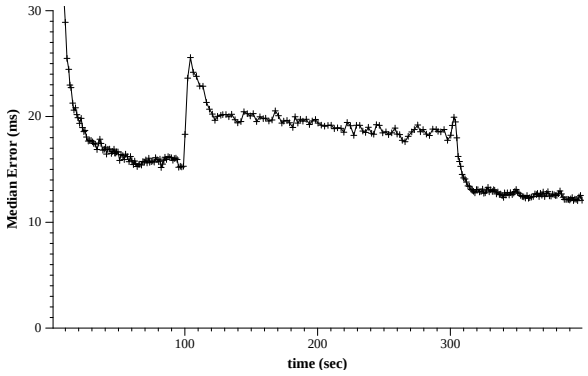

Figure 9: Vivaldi is able to adapt to changes in the network. In this experiment, we constructed a 100 node GTITM topology and allowed Vivaldi to determine coordinates for the nodes. The median error is plotted above against time. At time 100 one of the transit stub links is made 10 time larger; after around 20 seconds the system has reconverged to new coordinates. The error of the new system is larger in the original configuration. At time 300 the link goes back to its normal size and the system quickly reconverges to the original error.

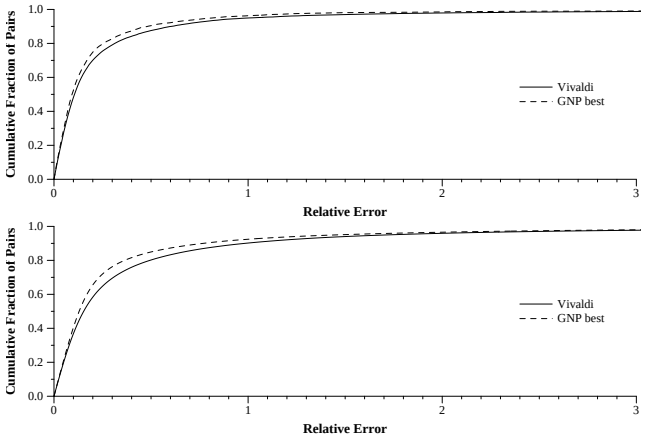

Figure 10: The cumulative distribution of prediction error for 2-dimensional Euclidean coordinates chosen by Vivaldi and GNP on the PlanetLab data set (top) and the King data set (bottom).

4.5 Accuracy

To evaluate Vivaldi's accuracy, we compared it against GNP [15], a centralized algorithm, on the PlanetLab and King data sets. Figure 10 compares the cumulative distribution of prediction error for the 2-dimensional Euclidean coordinates chosen by Vivaldi and GNP for both the PlanetLab and King data sets, using using 32 neighbors (Vivaldi) or landmarks (GNP). Vivaldi's error is competitive with that of GNP.

In Section 4.3 we discussed how Vivaldi can avoid "folding" the coordinate space by communicating with some distant nodes. We also find that neighbor selection affects accuracy in another way: preferentially collecting RTT samples from some nodes that are nearby in the network improves prediction accuracy. This was first demonstrated by PIC [3]. In the experiments presented in this section, each Vivaldi node took measurements from 16 nearby neighbors (found using the simulator's global knowledge of the network) and 16 random neighbors. Because GNP's performance depends on the choice of landmarks in the network, we performed 64 GNP

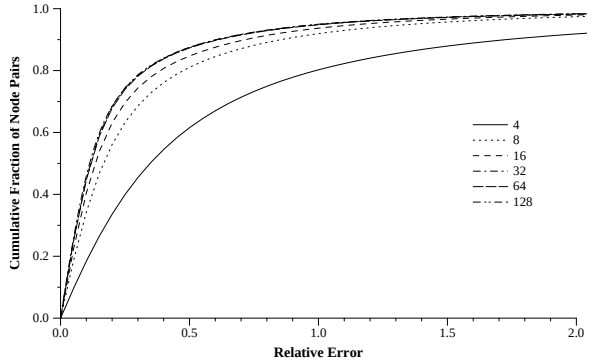

Figure 11: The cumulative distribution of prediction error for 3-dimensional coordinates chosen by Vivaldi using different numbers of neighbors.

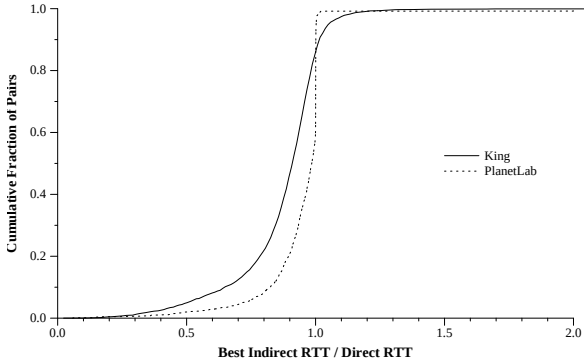

Figure 12: The cumulative distribution of the ratios of the RTTs of the best two-hop path to the direct path between each pair of King and PlanetLab nodes. The best indirect path usually has lower RTT than the direct path.

experiments with random landmark sets and chose the set that gave the lowest median error.

The number of neighbors also affects the accuracy of Vivaldi. Figure 11 shows the distribution of RTT prediction errors for varying numbers of neighbors using 3-dimensional coordinates. The neighbors were chosen using half nearby neighbors as described above. Vivaldi's performance increases rapidly until about 32 neighbors, after which time it does not improve much. GNP requires fewer neighbors than Vivaldi (it works well with around 16), but Vivaldi is less sensitive to neighbor placement and can use any node in the system as a neighbor.

5. MODEL SELECTION

Vivaldi was inspired by analogy to a real-world (and thus three-dimensional Euclidean) mass-spring system. As discussed in Section 2, the algorithm can use other coordinate spaces by redefining the coordinate subtraction, vector norm, and scalar multiplication operations. In this section, we consider a few possible coordinate spaces that might better capture the Internet's underlying structure for the King and PlanetLab data sets.

5.1 Triangle inequalities

Before considering any specific coordinate spaces, let us first consider how well we can expect to do. Almost any coordinate space we might consider satisfies the triangle inequality, which states that the distance directly between two nodes A and C should should be less than or equal to the distance along a path detouring from A to B and then to C. One should only expect to be able to find a coordinate space consistent with a set of inter-node latencies if the latencies themselves satisfy the triangle inequality.

Figure 12 presents an evaluation of the extent of such violations. For each pair of nodes, we found the lowest-RTT path through any intermediate node and calculated the ratio of the RTTs of the indirect and direct paths. Figure 12 plots the cumulative distribution of these ratios for the PlanetLab and King data sets. The vast majority of node pairs in the King data set are part of a triangle violation: these small violations are due mainly to measurement inaccuracy. A smaller number of severe violations are present in both datasets, as well as in the analysis of Tang and Crovella [29]. Because only around five percent of node pairs have a significantly shorter two-hop path, we expect that both data sets will be "embeddable" in a Euclidean space.

We also count the number of triples (i, j, k) that violate the constraint $|x_i - x_j| + |x_k - x_j| > |x_i - x_j| + \epsilon$. We include the constant term ϵ to avoid counting marginal violations (the error in measuring these links is likely on the order of several milliseconds). For $\epsilon = 5ms$, we find that 4.5% of the triples in the King data set violate the triangle inequality.

Of course, while a data set with many triangle inequalities will do poorly in most coordinate spaces we'd consider, a data set with few triangle inequalities still might not fit well into arbitrary coordinate spaces. The small number of triangle inequalities in the data only suggests that we are not doomed from the start.

5.2 Euclidean spaces

First, we explore the use of Euclidean coordinate spaces. These have the familiar equations:

$$\begin{aligned} [x_1, \cdots, x_n] - [y_1, \cdots, y_n] &= [x_1 - y_1, \cdots, x_n - y_n] \\ \|[x_1, \cdots, x_n]\| &= \sqrt{x_1^2 + \cdots + x_n^2} \\ \alpha \times [x_1, \cdots, x_n] &= [\alpha x_1, \cdots, \alpha x_n] \end{aligned}$$

If we choose to use a Euclidean coordinate space, the first question is how many dimensions to use. We use a principal components analysis as in Cox and Dabek [4] and Tang and Crovella [29], to characterize the dimensionality of Internet coordinates. The analysis suggests that the coordinates primarily use two to three dimensions, with little variation in the others. That is, insofar as the data is Euclidean, it is only two- or three-dimensional. This finding is a somewhat surprising result given the complexity of the Internet.

Examining the data reveals that the latencies in the data are dominated by geographic distance. If geographic distance were the only factor in latency, a 2-dimensional model would be sufficient. However, the fit is not perfect, probably due to aspects of the network like access-link delays and the fact that nodes often take inefficient routes as they move from one backbone to another (as happens, for example, in hot-potato routing). As we add more dimensions, the accuracy of the fit improves slightly, probably because the extra dimensions allow Vivaldi more "wiggle room" for placing hard-to-fit points.

Figure 13 plots the CDF of relative errors for 2-, 3-, and 5-dimensional Euclidean coordinates, for both the PlanetLab and King data sets. Adding extra dimensions past three does not make a significant improvement in the fit. While using coordinates with more dimensions does improve the quality of the fit, it also increases the communication overhead required to run Vivaldi; for this reason

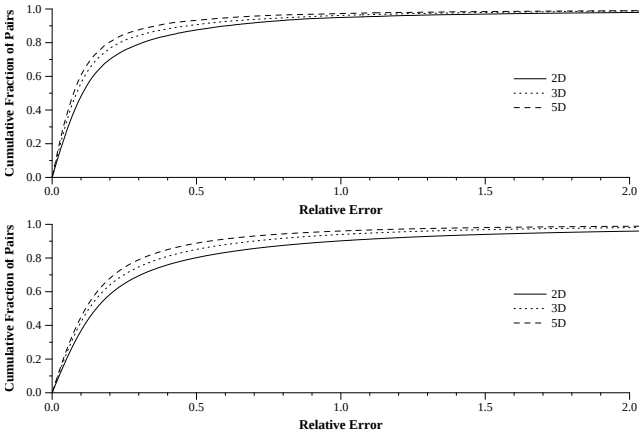

Figure 13: The cumulative distribution of Vivaldi's prediction error for various numbers of Euclidean dimensions for the PlanetLab (top) and King (bottom) data sets.

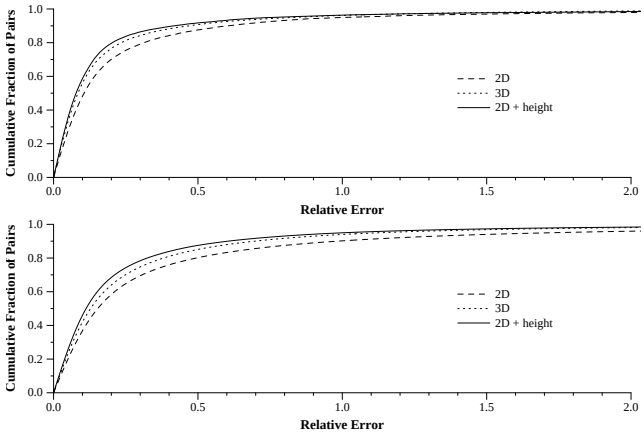

Figure 15: The cumulative distribution of prediction error for 2- and 3-dimensional Euclidean coordinates and height vectors chosen by Vivaldi for the PlanetLab (top) and King (bottom) data sets.

we prefer the lowest dimensional coordinates that allow for accurate predictions.

5.3 Spherical coordinates

Because we know that the distances we are attempting to model are drawn from paths along the surface of a sphere (namely the Earth), we might expect that a spherical distance function would provide a more accurate model.

We adapted Vivaldi to use spherical coordinates; instead of expressing a force vector in Euclidean space, a node moves some fraction of the angular distance towards or away from another node.

Figure 14 plots the CDF of relative errors for Vivaldi's choice 2-dimensional Euclidean coordinates as well as spherical coordinates with varying radii, for the PlanetLab and King data sets and a synthetic data set generated from a spherical model. In (a) and (b), the error is large until about 80 ms; before this, the sphere's surface is likely too small for the nodes to spread out.

The spherical model's error is similar to the Euclidean model's error and does not degrade as the radius is increased. This finding suggests that all points cluster on one side of the sphere as the sphere provides approximately $2\pi r^2$ of surface area, approximating a Euclidean plane in which to work. To test this hypothesis we tried Vivaldi on a synthetic network of nodes chosen to fit a sphere of radius 80 ms. The error for the spherical fit is zero when the radius of the modeled sphere is 80ms. It degrades as the radius increases beyond its optimal value. A fit to a 2-dimensional Euclidean space produced a larger error than the (near-perfect) spherical model on this data set.

We suspect that the underlying reason that spherical coordinates do not model the Internet well is that the paths through the Internet do not "wrap around" the Earth appreciably. Inspection of Internet paths originating in east Asia suggests that few links connect Asia and Europe directly. For instance, packets sent from Korea to Israel travel east across two oceans rather than west across land. Some paths do connect Asia and Europe directly, of course, but they are not prevalent in the data. A spherical model assumes that such links would always be used when they make the path shorter. Since this case is not the usual one, the fact that the spherical model correctly predicts the few paths across Asia is negated by the fact that it incorrectly predicts the many paths that go the long way, avoiding Asia. Anecdotal evidence gathered with traceroute on PlanetLab nodes supports this observation.

5.4 Height vectors

A height vector consists of a Euclidean coordinate augmented with a height. The Euclidean portion models a high-speed Internet core with latencies proportional to geographic distance, while the height models the time it takes packets to travel the access link from the node to the core. The cause of the access link latency may be queuing delay (as in the case of an oversubscribed cable line), low bandwidth (as in the case of DSL, cable modems, or telephone modems), or even the sheer length of the link (as in the case of long-distance fiber-optic cables).

A packet sent from one node to another must travel the source node's height, then travel in the Euclidean space, then travel the destination node's height. Even if the two nodes have the same height, the distance between them is their Euclidean distance plus the two heights. This is the fundamental difference between height vectors and adding a dimension to the Euclidean space. Intuitively, packet transmission can only be done in the core, not above it.

The height vector model is implemented by redefining the usual vector operations (note the + on the right hand side of the subtraction equation):

$$\begin{aligned} [x, x_h] - [y, y_h] &= [(x - y), x_h + y_h] \\ \left\|[x, x_h]\right\| &= \left\|x\right\| + x_h \\ \alpha \times [x, x_h] &= [\alpha x, \alpha x_h] \end{aligned}$$

Each node has a positive height element in its coordinates, so that its height can always be scaled up or down.

The effect of these equations is the following. In a normal Euclidean space, a node that finds itself too close to another node will move away from the other node. A node that finds itself too close to nodes on all sides has nowhere to go: the spring forces cancel out and it remains where it is. In the height vector system, the forces cancel out in the Euclidean plane, but the height forces reinforce each other, pushing the node up away from the Euclidean plane. Similarly, a node that finds itself too far away from other nodes on all sides will move down closer to the plane.

Figure 15 shows that height vectors perform better than both 2D and 3D Euclidean coordinates.

Examination of the coordinates that the height vector model assigns to hosts in the PlanetLab data set shows that the model captures the effect we hoped. Well-connected nodes, such as the ones at New York University, are assigned the minimum height. Two

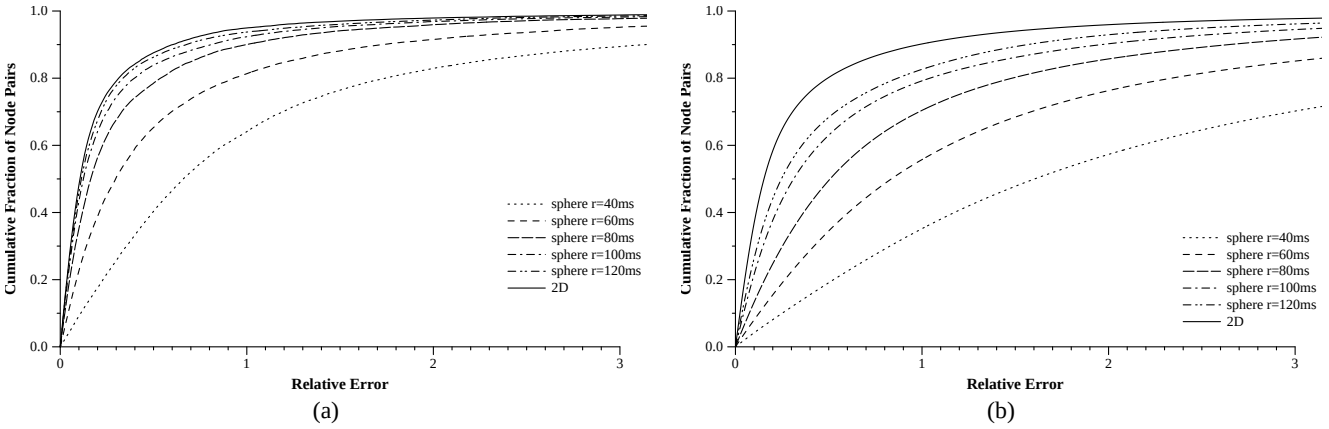

Figure 14: The cumulative distribution of prediction error for spherical coordinates of various radii chosen by Vivaldi for the (a) PlanetLab and (b) King data sets.

Brazilian nodes are assigned coordinates at approximately 95 ms above the United States. Using traceroute on these nodes we see that 95 ms is approximately the distance to the nodes' connection to the backbone of the Internet2 network inside the United States. Because the Brazilian nodes send their traffic via the United States to other nodes in this small data set they are best represented as 95 ms "above" the continental U.S. If there were a larger variety of Brazilian nodes and links in the data set, the nodes would be given their own region of the Euclidean plane.

5.5 Graphical comparison

The plots in Figure 16 show the final placement of nodes in two and three dimensional Euclidean space and with height vectors. The Euclidean plots have a cloud of nodes that are far away from everyone and don't seem to know where to go. These nodes are also the bulk of the ones with large errors. The height plots have a place for these nodes: high above the rest of the network. This results in a smaller maximum node error for the height plots, as well as a smaller median error. The smaller errors suggest that the height vectors are a more accurate reflection of the structure of the Internet for the nodes in the data sets.

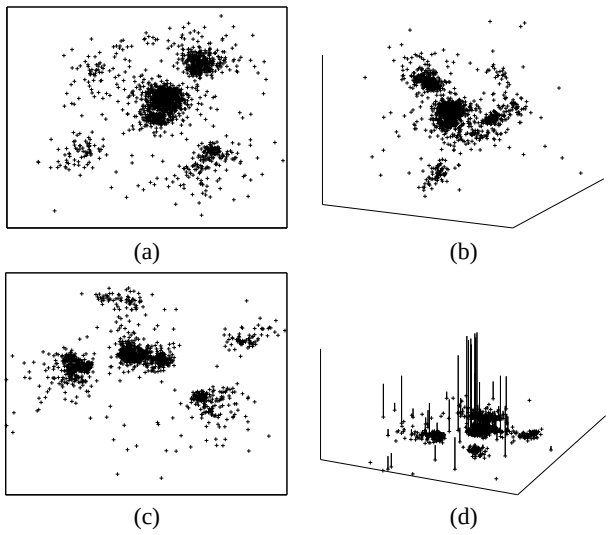

Figure 16: The node placement chosen by Vivaldi for the King data set (a) in two dimensions, (b) in three dimensions, (c) with height vectors projected onto the *xy* plane, and (d) with height vectors rotated to show the heights. The "fringe nodes" in (a) and (b) are not as prevalent in (c).

6. RELATED WORK

A number of systems have been proposed for computing synthetic coordinates. In addition, the mass-spring minimization used by Vivaldi has also been used previously. Unlike any one of the related systems, Vivaldi is fully decentralized, targets the Internet, and adapts well to changes in the network. Vivaldi uses height vectors to improve its latency predictions, while previous systems mostly use Euclidean coordinates.

6.1 Centralized coordinate systems

Many existing synthetic coordinate systems involve a centralized component (such as a set of landmark nodes) or require global knowledge of node measurements. Vivaldi's main contribution relative to these systems is that it is decentralized and does not require a fixed infrastructure.

Vivaldi was inspired by GNP [17], which demonstrated the possibility of calculating synthetic coordinates to predict Internet latencies. GNP relies on a small number (5-20) of "landmark" nodes; other nodes choose coordinates based on RTT measurements to the landmarks. The choice of landmarks significantly affects the accuracy of GNP's RTT predictions. Requiring that certain nodes be designated as landmarks may be a burden on symmetric systems (such as peer-to-peer systems). In addition, landmark systems, unlike Vivaldi, do not take advantage of all communication between nodes: only measurements to landmarks are helpful in updating coordinates.

NPS [16] is a version of GNP which addresses system-level issues involved in deploying a coordinate system. NPS includes a hierarchical system for reducing load on the landmark nodes, a strategy for mitigating the effects of malicious nodes, a congestion control mechanism , and a work-around for NATs. NPS also demonstrates how a landmark-based coordinate system can adapt to changing network conditions by periodically communicating with the landmark nodes and recomputing coordinates.

Lighthouse [19] is an extension of GNP that is intended to be more scalable. Lighthouse, like GNP, has a special set of landmark nodes. A node that joins Lighthouse does not have to query those global landmarks. Instead, the new node can query any existing

set of nodes to find its coordinates relative to that set, and then transform those coordinates into coordinates relative to the global landmarks.

Tang and Crovella propose a coordinate system based on "virtual" landmarks [29]; this scheme is more computationally efficient than GNP and is not as dependent on the positions of the landmarks. The authors also present an analysis of the dimensionality of their measured latencies which suggests that low-dimensional coordinates can effectively model Internet latencies. This result agrees with the results presented in Section 5

6.2 Decentralized Internet coordinate systems

PIC [3], like Vivaldi, can use any node that knows its coordinates as a landmark for future nodes: PIC does not require explicitly designated landmarks. A PIC node either knows or doesn't know its coordinates. Vivaldi incorporates a continuous measure of how certain a node is about its coordinates; this approach helps adapt to changing network conditions or network partitions. PIC does not seem to be suited to very dynamic conditions: it runs the Simplex minimization procedure completely after a node measures the latencies to the landmarks. This makes PIC react very quickly to new measurements. In building Vivaldi we found that reacting too quickly to measurements (using a large time-step) caused the coordinates to oscillate.

PIC defends the security of its coordinate system against malicious participants using a test based on the triangle inequality. Vivaldi defends against high-error nodes, but not malicious nodes. PIC uses an active node discovery protocol to find a set of nearby nodes to use in computing coordinates. In contrast, Vivaldi piggybacks on application-level traffic, assuming that applications that care about coordinate accuracy to nearby nodes will contact them anyway.

NPS [16] uses a decentralized algorithm. Instead of sending measurements to a central node that runs the Simplex algorithm to determine landmark coordinates (as GNP does), each NPS landmark re-runs the minimization itself each time it measures the latency to a new node. Vivaldi operates in much the same manner, but uses a mass-spring system instead of the Simplex algorithm to perform the minimization.

6.3 Coordinate systems for wireless nets

The distributed systems described in this section target wireless networks where distance, either in network latency or physical separation, reflects geographic proximity fairly closely. Vivaldi is intended for use in the Internet, where the topology of the network is much less obvious *a priori*.

AFL [21], a distributed node localization system for Cricket [22] sensors, uses spring relaxation. The Cricket sensors use ultrasound propagation times to measure inter-sensor distances and cooperate to derive coordinates consistent with those distances. Most of the complexity of AFL is dedicated to solving a problem that doesn't affect Vivaldi: preventing the coordinate system from "folding" over itself. Unlike the sensors used in the Cricket project, we assume that Vivaldi hosts can measure the latency to distant nodes; this eliminates the folding problem.

Other systems (such as ABC [25]) operate by propagating known coordinates from a fixed set of anchor nodes in the core of the network. Each node can then find its location in terms of the propogated coordinates using techniques like triangulation.

Rao *et. al.* [23] describe an algorithm for computing virtual coordinates to enable geographic forwarding in a wireless ad-hoc network. Their algorithm does not attempt to predict latencies; instead, the purpose is to make sure that directional routing works.

6.4 Spring relaxation

Several systems use spring relaxation to find minimal energy configurations. Vivaldi's use of spring relaxation was inspired by an algorithm to recover a three dimensional model of a surface from a set of unorganized points described by Hugues Hoppe [11]. Hoppe introduces spring forces to guide the optimization of a reconstructed surface.

Mogul describes a spring relaxation algorithm to aid in the visualization of traffic patterns on local area networks [14]. Spring relaxation is used to produce a 2-dimensional representation of the graph of traffic patterns; the results of the relaxation are not used to predict latencies or traffic patterns.

The Mithos [24, 30] overlay network uses a spring relaxation technique to assign location-aware IDs to nodes.

Shavitt and Tankel's Big Bang system [26] simulates a potential field similar to Vivaldi's mass-spring system. Their simulation models each particle's momentum explicitly and then introduces friction in order to cause the simulation to converge to a stable state. Vivaldi accomplishes the same effect by ignoring the kinetic energy of the springs. The Big Bang system is more complicated than Vivaldi and seems to require global knowledge of the system; it is not clear to us how to decentralize it.

6.5 Internet models

Vivaldi improves its latency predictions with a new coordinate space that places nodes some distance "above" a Euclidean space. Previous synthetic coordinate systems have concentrated on pure Euclidean spaces or other simple geometric spaces like the surfaces of spheres and tori.

Shavitt and Tankel [27] recently proposed using a hyperbolic coordinate space to model the Internet. Conceptually the height vectors can be thought of as a rough approximation of hyperbolic spaces. The hyperbolic model may address a shortcoming of the height model; the height model implicitly assumes that each node is behind its own access link. If two nodes are behind the same high-latency access link, the height model will incorrectly predict a large latency between the two nodes: the distance down to the plane and back up. Comparing the height vectors and hyperbolic model directly to determine which is a better model for the Internet is future work.

6.6 Other location techniques

IDMaps [8] is an infrastructure to help hosts predict Internet RTT to other hosts. The infrastructure consists of a few hundred or thousand *tracer* nodes. Every tracer measures the Internet RTT to every other tracer. The tracers also measure the RTT to every CIDR address prefix, and jointly determine which tracer is closest to each prefix. Then the RTT between host h_1 and host h_2 can be estimated as the RTT from the prefix of h_1 to that prefix's tracer, plus the RTT from the prefix of h_2 to that prefix's tracer, plus the RTT between the two tracers. An advantage of IDMaps over Vivaldi is that IDMaps reasons about IP address prefixes, so it can make predictions about hosts that are not aware of the IDMaps system.

The IP2Geo system [18] estimates the physical location of a remote server using information from the content of DNS names, whois queries, pings from fixed locations, and BGP information. IP2Geo differs from Vivaldi mainly in that it attempts to predict physical location rather than network latency.

7. CONCLUSIONS AND FUTURE WORK

Vivaldi is a simple, adaptive, decentralized algorithm for computing synthetic coordinates for Internet hosts. Vivaldi requires no

fixed infrastructure, supports a wide range of communication patterns, and is able to piggy-back network sampling on application traffic. Vivaldi includes refinements that adaptively tune its time step parameter to cause the system to converge to accurate solutions quickly and to maintain accuracy even as large numbers of new hosts join the network.

By evaluating the performance of Vivaldi on a large data set generated from measurements of Internet hosts, we have investigated the extent to which the Internet can be represented in simple geometric spaces. We propose a new model, height vectors, which should be of use to all coordinate algorithms. Attempting to understand characteristics of the Internet by studying the way it is modeled by various geometric spaces is a promising line of future research.

Because Vivaldi requires no infrastructure and is simple to implement, it is easy to deploy in existing applications. We modified DHash, a distributed hash table, to take advantage of Vivaldi and reduced time required to fetch a block in DHash by 40% on a global test-bed [7]. We hope that Vivaldi's simplicity will allow other distributed systems to adopt it.

Acknowledgments

This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660. Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

REFERENCES

[1] BitTorrent. http://bitconjurer.org/BitTorrent/protocol.html.

[2] K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling Internet topology. *IEEE Communications*, 35(6):160–163, June 1997.

[3] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In *International Conference on Distributed Systems*, Tokyo, Japan, March 2004.

[4] R. Cox and F. Dabek. Learning Euclidean coordinates for Internet hosts. http://pdos.lcs.mit.edu/~rsc/6867.pdf, December 2002.

[5] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–205, Oct. 2001.

[7] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[8] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking*, Oct. 2001.

[9] T. Gil, J. Li, F. Kaashoek, and R. Morris. Peer-to-peer simulator, 2003. http://pdos.lcs.mit.edu/p2psim.

[10] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proc. of SIGCOMM IMW 2002*, pages 5–18, November 2002.

[11] H. Hoppe. *Surface reconstruction from unorganized points*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

[12] KaZaA media dekstop. `http://www.kazaa.com/`.

[13] P. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *Proc. ACM SIGCOMM*, pages 123–133, Stanford, CA, 1988.

[14] J. C. Mogul. Efficient use of workstations for passive monitoring of local area networks. Research Report 90/5, Digital Western Research Laboratory, July 1990.

[15] E. Ng. GNP software, 2003. `http://www-2.cs.cmu.edu/~eugeneng/research/gnp/software.html`.

[16] T. E. Ng and H. Zhang. A network positioning system for the Internet. In *Proc. USENIX Conference*, June 2004.

[17] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proceedings of IEEE Infocom*, pages 170–179, 2002.

[18] V. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *Proc. ACM SIGCOMM*, pages 173–185, San Diego, Aug. 2001.

[19] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In *IPTPS*, 2003.

[20] Planetlab. www.planet-lab.org.

[21] N. Priyantha, H. Balakrishnan, E. Demaine, and S. Teller. Anchor-free distributed localization in sensor networks. Technical Report TR-892, MIT LCS, Apr. 2003.

[22] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Proc. 6th ACM MOBICOM Conf.*, Boston, MA, Aug. 2000.

[23] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *ACM MobiCom Conference*, pages 96 – 108, Sept. 2003.

[24] R. Rinaldi and M. Waldvogel. Routing and data location in overlay peer-to-peer networks. Research Report RZ–3433, IBM, July 2002.

[25] C. Savarese, J. M. Rabaey, and J. Beutel. Locationing in distributed ad-hoc wireless sensor networks. In *ICASSP*, pages 2037–2040, May 2001.

[26] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in Euclidean space. In *Proc. of IEEE Infocom*, April 2003.

[27] Y. Shavitt and T. Tankel. On the curvature of the Internet and its usage for overlay construction and distance estimation. In *Proc. of IEEE Infocom*, April 2004.

[28] J. Stribling. All-pairs-ping trace of PlanetLab, 2004. http://pdos.lcs.mit.edu/ strib/.

[29] L. Tang and M. Crovella. Virtual landmarks for the Internet. In *Internet Measurement Conference*, pages 143 – 152, Miami Beach, FL, October 2003.

[30] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. In *Hotnets-I*, 2002.

[31] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA USA, December 2002.