Stable and Accurate Network Coordinates

Jonathan Ledlie, Peter Pietzuch, and Margo Seltzer
Division of Engineering and Applied Science
Harvard University, Cambridge, MA, USA

Abstract

Network coordinates provide a scalable way to estimate latencies among large numbers of hosts. While there are several algorithms for producing coordinates, none account for the fact that nodes observe a stream of distinct observations that may vary by as much as three orders-of-magnitude. With such variable data, coordinate systems are prone to high error and instability in live deployments. In addition, dynamics such as triangle violations can lead to coordinate oscillations, producing further instability and making it difficult for applications to know when their coordinates have truly changed. Because simulation results demonstrate that network coordinates are capable of providing low cost and sufficiently accurate answers to common queries, it is vital that we develop the ability to obtain similar results in practice. We propose two filters which combined to improve network coordinate accuracy by 54% *and coordinate stability by* 96% *when run on a real, large-scale network.*

1 Introduction

Decentralized network coordinate algorithms take inter-node latencies and embed them in a relative coordinate space [5, 7, 15, 16, 18, 24, 25, 27]. With appropriate input, they enable latency prediction among nodes that have never communicated and allow complex distributed systems problems to be solved geometrically. They are useful in a wide range of contexts, including large-scale content distribution, routing, and storage. In particular, network coordinates (NCs) are an essential component of our work on distributed database query optimization [20] and of Abraham's Compact Routing [1] and are also starting to be used by one of the most popular BitTorrent clients [2].

Network coordinates are a powerful abstraction with attractive properties: they have low overhead because they can interpolate non-existing measurements; their embedding error is sufficiently low for practical applications; and the trade-off between measurement overhead and accuracy is explicit by adjusting their observation frequency. Their main power is that they provide a rich array of geometric primitives for solving distributed systems problems, such as nearest web cache selection, content distribution, and efficient placement of resources [19]. For example, a web cache can be placed at the centroid of all of the clients' coordinates accessing the cache. The low dimensionality of NCs makes a wide range of algorithms from computational geometry applicable to networking problems. Their geometric interpretation also helps unify wired and wireless networks, making similar algorithms applicable to both domains and thus simplifying the design of heterogeneous systems.

Network coordinate schemes have, as yet, only performed well in simulation. When run on a live system, the basic algorithms do not produce stable, accurate coordinates. The discrepancies are primarily a result of (a) the orders-of-magnitude variation in latency measurements between the same pairs of nodes that occur when running NCs on a real network and (b) the inherent impossibility of latencies to be perfectly embedded when triangle inequality violations exist, causing oscillations. The simulation studies have used a derived latency matrix, typically containing the median values for links measured over hours or days. In this paper, we describe how the addition of two types of filters produces coordinates that are stable, accurate, and adaptive to changing network conditions. As a result, our techniques yield high-quality NCs under "real world" conditions.

In the next section, we introduce algorithms for computing NCs and show how to measure their quality, emphasizing a new metric called *stability* and its importance for applications. In Section 3, we examine a latency distribution that exemplifies a typical input and discuss why NC algorithms experience difficulty when used without a static latency matrix. In Section 4, we present a simple method for stabilizing coordinates by keeping a small history of samples associated with each link. These *latency filters* improve both coordinate stability and accuracy; however, coordinate stability remains at a level unacceptable to most applications. In Section 5, we differentiate between application- and system-level coordinates and compare four heuristics for improving application-level stability while maintaining

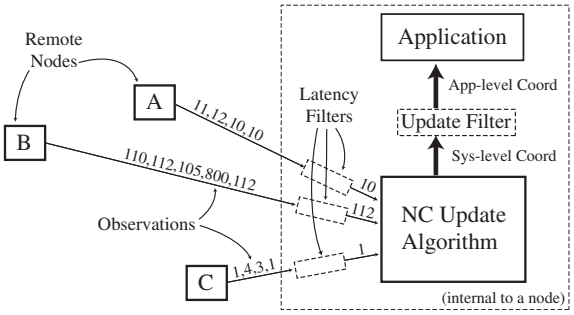

Figure 1. Latency and Update filters

accuracy. We find that using a sliding window for change detection as an *update filter* allows an application's view of its network coordinates to become significantly more stable. In Section 6, we build histories and application-level coordinates into an implementation that we run on a large network, resulting in a 54% improvement in accuracy and a 96% improvement coordinate stability. As Figure 1 illustrates, this paper examines (a) what kinds of *latency filters* turn latency observations into useful network coordinates and (b) what kinds of *update filters* notify applications only with significant coordinate changes. In Section 7, we discuss related work; we conclude in Section 8.

2 Network Coordinate Algorithms

All NC algorithms embed a characteristic of a network, typically latency, into a metric space, but the methods used differ along several axes. There are two fundamental differentiating characteristics and two that are somewhat secondary. The geometry used for the metric space is fundamental: while most work uses low dimensional Euclidean spaces, Shavitt and Tankel propose a hyperbolic embedding [25], and Tang and Crovella examine Lipschitz coordinates [27]. A second fundamental characteristic is whether all nodes are treated equally: whether there is a set of landmarks that serve as reference points (and generally perform an intensive set of measurements *a priori*). Two secondary characteristics are periodicity and the existence of a decentralized implementation. Costa's PIC is described as solving once for a node's coordinate, but its coordinates could be recomputed periodically. Finally, while the existence of a public decentralized implementation has helped the popularity of Vivaldi, for example, such an implementation is possible for most NC algorithms.

With a working distributed implementation, these different types of NCs offer, to varying degrees, three main benefits. First, two nodes do not need to have communicated previously for the latency between them to be estimated. Because all-to-all communication is not necessary, NCs scale to thousands or millions of nodes. Second, NCs can continue to refine coordinates as the true network conditions change over time. For example, if the latency of a link changes due to a BGP route change, coordinates can adjust and restabilize quickly. Lastly, NCs provide distributed systems with the ability to solve problems geometrically. For example, a node can learn of its approximate nearest neighbor without ever communicating with it, using background gossip of coordinates instead. Of course, NCs are not a panacea: any embedding of network latencies that includes triangle violations – and almost all non-trivial networks do – must induce a level of error. However, for the broad collection of applications that can use good approximations to their distributed problems, NCs are a worthy solution.

For purposes of presentation and evaluation, we use *Vivaldi* as the canonical example of a NC update algorithm. Because it has a simple, public, distributed algorithm, it is used in several projects including Bamboo [23], SBON [20], and Azureus [2]. However, these three projects have all experienced difficulties with coordinate accuracy and stability when run on real networks. Because all embedding methods require latency estimates as inputs and produce coordinates as outputs, our techniques should be directly applicable to them as well.

2.1 Vivaldi

Vivaldi models the network as a collection of springs that pull on each node's coordinate. Each node retains its coordinate $\overrightarrow{x_i}$ and its confidence in this coordinate $w_i \in (0, 1)$. All coordinates are the same low dimension, which is fixed *a priori*. Nodes adjust their coordinates and confidences through observations of their latencies to other nodes in the system. These observations can be explicit pings or may be gleaned from existing traffic. Through successive samples, each node refines its coordinates and increases its confidence. Like a network of springs, coordinates become more accurate and stable with each successive adjustment.

Each node updates its coordinate and confidence with each new latency observation based on the pseudocode shown in Figure 2. An observation consists of the remote node's coordinate $\overrightarrow{x_j}$, its confidence w_j, and a new measurement of the latency between the two nodes, i and j, l_{ij}. First, a weight w_s is assigned to this observation based on how confident nodes i and j are relative to one another (Line 1). In essence, this allows more confident nodes to tug harder than less confident ones. Second, they find how far off the observation was from what was expected based on the coordinates; this is the relative error ϵ of this measurement (Line 2). Third, node i updates its confidence w_i with an exponentially-weighted moving average (EWMA). Unlike most EWMAs, however, the α, or weight given to the current observation, is not fixed. Instead it is weighted according to the trustworthiness of the current observation (Lines 3-4). If this causes node i's confidence to go above

VIVALDI($l_{ij}, \overrightarrow{x_j}, w_j$)
1 $w_s = \frac{w_i}{w_i + w_j}$
2 $\epsilon = \frac{|\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}|}{l_{ij}}$
3 $\alpha = c_e \times w_s$
4 $w_i = (\alpha \times \epsilon) + ((1 - \alpha) \times w_i)$
5 $\delta = c_c \times w_s$
6 $\overrightarrow{x_i} = \overrightarrow{x_i} + \delta \times (\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}) \times u(\overrightarrow{x_i} - \overrightarrow{x_j})$

Figure 2. *Vivaldi* **update algorithm**

one or below zero, it is forced to remain in bounds (not shown). Confidence is greatest when $w_i = 0$. Lines 5-6 update the coordinate. First, we compute this observation's pull on the coordinate, δ, based on i and j's confidence. Then, in line 6, δ dampens the magnitude and direction (normalized with u, the unit vector function) of the change applied to the coordinate. Constants c_e and c_c affect the maximum change an observation can exert on confidence and coordinates, respectively. They have the same effect as the tuning parameter in a standard EWMA: a low value of 0.05, for example, limits the weight given to any new observation and a high value of 0.25, for example, causes faster adjustments to new observations. Larger values for α may weigh outliers too heavily. We found any setting of c_c and c_e in this range to have minimal impact on large scale behavior.

Bootstrapping the algorithm is simple. We set all coordinates to the origin. Each node stores a list of *neighbors*, *i.e.,* nodes that it samples. We assume that a node knows at least one other node when it enters the system. In our implementation, nodes learn new neighbors by attaching the address of one other node to each sampling message, *i.e.,* through gossip, and sample their neighbors in round-robin order.

Vivaldi can be modified to include a *height* h, which changes the distance between nodes i, j to $\|\overrightarrow{x_i} - \overrightarrow{x_j}\| + h_i + h_j$. The purpose of *height* is to capture the latency of the access link, while the coordinates themselves capture the long-haul links. Because the growing body of projects using Vivaldi has not used height, we did not include it, although the techniques we present would allow for its use. We present results using three dimensions.

2.2 Measuring Coordinate Systems

Comparing the difference between the expected and actual latencies for an observation measures accuracy. The error of a link for a particular observation l_{ij} is:

$$e = | \ \|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij} \ |$$

Depending on context, the accuracy for the system is the sum of these quantities for all nodes, the sum of the error squared (the mean squared error), or the median for each node. Accuracy can also be normalized by dividing by l_{ij}; this *relative error* is ϵ in Figure 2. We use relative error as the accuracy metric because it facilitates comparison of a wide range of latencies.

Recently, Lua *et al.* proposed NC error metrics that better capture application impact [13]. *Relative rank loss* (rrl) determines how well a network coordinate scheme respects the relative ordering of all pairs of neighbors. Thus, for each node x, if $(d_{xi} > d_{xj} \wedge l_{xi} < l_{xj})$ or $(d_{xi} < d_{xj} \wedge l_{xi} > l_{xj})$, then the distances d between coordinates led to an incorrect prediction of the relative latencies l. This metric is important for applications that make decisions dependent on the relative ordering of nodes, for example, when updating routing table entries. In related work [21], we extended rrl to capture the *magnitude* of each rank misordering as well: swapping the rank of two nodes that are 1ms apart is less severe than reordering nodes that are 100ms apart. This *weighted rrl* (wrrl) is computed by taking the sum of the latency penalties l_{ij} of node pairs ranked incorrectly, normalized over the worst case latency penalty. Finally, the *relative application latency penalty* (ralp) expresses the percentage of additional latency that an application will notice when using network coordinates for rank ordering. It is approximated by summing the relative penalty l_{ij}/l_{xi} for all pairs that are incorrectly ranked. In Section 4.1, we evaluate our filtering techniques with these application metrics and show that the results are consistent with the behavior exhibited by the relative error.

We measure *per-node* relative error instead of *per-link* relative error. The distribution of per-node relative error is the collection of errors for each node for all of its observations. Measuring per-link error assumes that a static, scalar latency matrix exists against which we could compare coordinates after a number of iterations. Because our underlying network is changing, this matrix, and hence this metric, cannot be computed.

Stable coordinates are particularly important when an application uses network coordinates and a coordinate change triggers application activity. A stable coordinate system is one in which coordinates are not changing over time, assuming that the network itself is unchanging. Thus, links may produce some distribution of observations, but as long as this distribution does not change, neither should stabilized coordinates. We use the rate of coordinate change

$$s = \frac{\sum \Delta \overrightarrow{x_i}}{t}$$

to quantify stability. In our metric space, the numerator is in milliseconds and we measure change in this space in seconds; thus, stability is in ms/sec unless otherwise noted.

3 Latency Measurements

When we first implemented NCs, we found that lone samples, orders-of-magnitude greater than expected, peri-

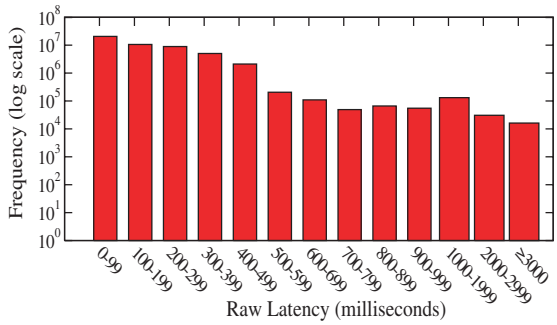

Figure 3. Unfiltered latencies (all nodes)

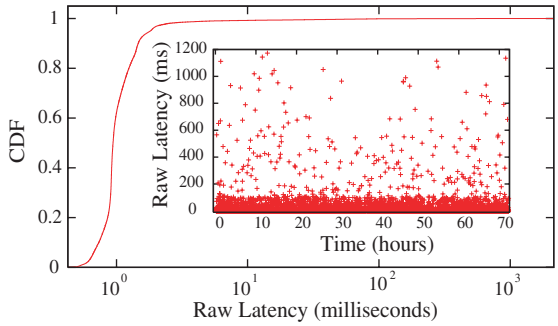

Figure 4. Unfiltered latencies (two nodes)

odically distorted the entire coordinate system. These instabilities appeared when the algorithm used raw latency data.

Raw latency data show rare but persistent samples orders-of-magnitude larger than the common case. We collected a set of latency data from 269 PlanetLab [17] nodes over three days starting May 2, 2005, producing 43 million samples. PlanetLab is a collection of approximately 500 machines spread around the world, located primarily at universities and research labs. To gather the trace, each node measured the latency to another node with an application-level UDP ping once per second.

We summarize the total distribution of measurements in Figure 3. The data show that 0.4% of the measurements are greater than one second, which is longer than the common case even for inter-continental links. Instead of a steady stream of measurements, the fact that many measurements are above the largest expected latency suggests that many links may be experiencing serious delays that NCs must automatically incorporate. The broad range of measurements severely curtails accuracy and stability.

We examined individual links to confirm that they too exhibited similar behavior. Not only did the entire distribution have a long tail, with most links below several hundred milliseconds, but individual links had as well. Figure 4 illustrates one representative link. It shows that some observations extend beyond the median and that these infrequent order-of-magnitude delays are spread over time.

Because of the long tail, the mean of the raw values would not be a good predictor for future observations. Instead, the expected latency appeared to be predictable by taking a low percentile of some portion of the previous observations. This expected latency is a better measure of what NCs should use as its approximation of the link latency, not the raw values. By giving NCs a steadier input that is able to predict subsequent values with high accuracy, each link should experience lower relative error and greater stability by exhibiting less coordinate change over time.

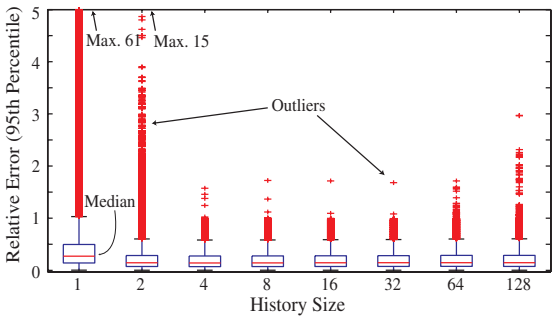

Figure 5. MP filter prediction error

4 Filtering with Histories

Based on our analysis of link latencies, a percentile of some window of previous observations appeared to be a good predictor of future values. Statistically, this is known as a Moving Percentile (MP) filter, a variant on the Moving Median filter, and has been used to filter out heavy-tailed error in other disciplines [8, 14]. It is a non-linear filter, which removes non-Gaussian noise and lets through low frequencies. MP filters exhibit edge preservation and are robust against noise impulses. A MP filter has two parameters: (1) the size h of the history window and (2) the percentile p returned as the prediction for the next observation.

To examine the predictive effectiveness of the MP filter with different parameters, we examined how the filter performed on each link from the PlanetLab trace. Each link consists of a series of observations; the relative error is the difference between the filter's prediction and the next observation divided by the next observation.

We ran an experiment in which we varied the size of the window and the percentile used to predict the next value. Using the three day trace, we applied different filters to predict what the next observation would be and calculated the relative error between each prediction and the true observation. Figure 5 shows filters' abilities to predict the next latency for each link as we vary the history size h and keep $p = 25$. The results show that short histories, *e.g.,* only four observations, achieve the best performance (lowest error) with the fewest outliers. Using $p = 25$ resulted in slightly lower error than $p = 50$ for the MP filter.

Although long histories do not perform substantially worse, intuitively it makes sense that longer histories do not

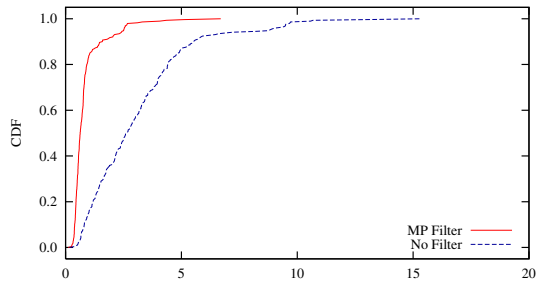

Figure 6. 95^{th} Percentile Relative Error

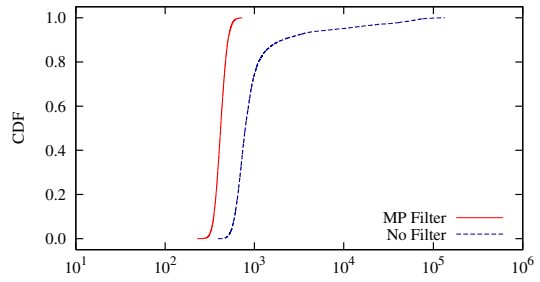

Figure 7. Instability (log scale)

perform better: they are slow to adjust to any changes in network conditions. That short histories perform well is good for three reasons: (1) they can be acquired through fewer rounds of observations, (2) they require less state, and (3) they will be quickest to adjust to any latency shifts.

In the previous experiment, we made the assumption that the magnitude of the long tail behavior of latency measurements remains unchanged over time. In practice, this may not be the case because the long tail is caused by artifacts, such as security policies implemented by routers and temporary route changes due to unstable BGP routing policies, which are themselves dynamic in nature. These changes may affect the efficiency of the chosen p and h parameters over time. However, changes to the magnitude of the long tail occur at larger timescales, which we have not seen during our experiments so far. An adaptive solution would revisit the choice of p and h periodically to ensure that the filter remains a good predictor for future measurements.

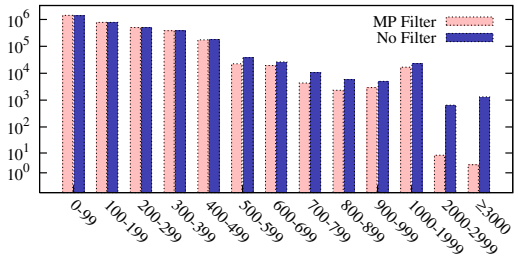

Figure 8. Filtered vs. Raw Latency (ms)

4.1 NCs with the Latency Filter

In order to compare NCs with and without the MP filter, we built a simulator that accepted our raw ping trace as input and mimicked the distributed behavior of Vivaldi. Through a comparison of running NCs on a real network and in our simulator, we found the simulator provided a high degree of verisimilitude.

Using the MP filter substantially improves both the accuracy and stability metrics. With the parameters that showed the best ability to predict subsequent samples — taking the 25^{th} percentile of the previous four observations (*i.e.,* the minimum) — we compared NCs with and without MP filtering. We ran NCs on a four hour section of the trace and show cumulative distributions for the second half of the run, eliminating start-up effects (we examine the rate of start-up in Section 5). We measured per-node accuracy and system-wide stability and summarize the results in Figures 6 and 7. The data show that the MP filter at least doubles accuracy and stability for most nodes. Its primary benefit, however, is that it eliminates the periodic distortion of the entire coordinate space that occurs with no filtering. This is shown through the reduction of the long tail of instability by three orders-of-magnitude. In the application we developed, these distortions cause a cascade of other updates to occur; using the MP filter ameliorated this problem substantially.

We also evaluated the efficiency of the MP filter in terms of the application metrics capturing rank loss described in Section 2.2. We calculated the rrl, wrrl, and ralp metrics for all the nodes in our set. The results show that the MP filter improves the 80^{th} percentile of rrl by 70%, of wrrl by 44%, and of ralp by 67% [21]. We conclude from this that applications using NCs to order nodes according to latency directly benefit from the MP filter.

Before turning to the non-linear MP filter, we considered two methods that are commonly used to smooth out measurement error, thresholds and exponential averaging, and a *confidence building* method specific to Vivaldi. Contrary to our initial expectations, these methods had negligible impact on accuracy or stability, and made conditions worse in some circumstances. An extended version of this paper presents details on these other methods [12].

5 Updating Application-level Coordinates

Violations of the triangle inequality with respect to RTT measurements have been shown to be a common occurrence on the Internet due to Internet routing policies. A recent study found around 18% triangle violations between 399 PlanetLab nodes [32]. Because of triangle violations, any NC algorithm that refines its coordinate periodically, especially with every observation, will produce coordinates

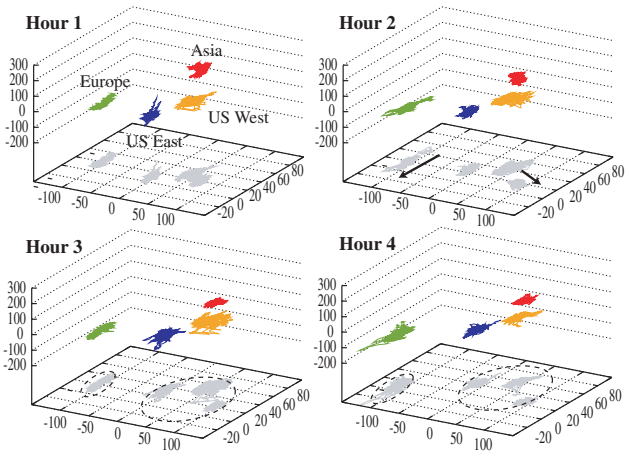

Figure 9. Coordinate change over time

that oscillate in a region – decreasing stability – with the size of that region dependent on the size of the violation.

Using a latency filter greatly improved stability and accuracy for a set of network coordinates. As Figure 7 showed, use of the filter clipped the heavy tail of instability. However, the system's coordinates are still changing at about $500ms/sec$. For an application using network coordinates, is all this movement necessary? Instead of being notified about slight changes in coordinates with every observation, most applications would prefer to be notified only when a *significant* change occurs. By designing the coordinate subsystem as a black box that only signals when there is significant change, we can limit application updates that, in turn, limit unnecessary application-level work. In our distributed database optimizer, for example, a coordinate change could initiate a cascade of events, culminating in one or more heavyweight process migrations. If the systems' coordinates have not changed significantly, there is no reason to begin this process. Of course, some applications would prefer a constant update: the subsystem should output both a system-level coordinate, $\overrightarrow{c_s}$, and an application-level one, $\overrightarrow{c_a}$.

Before considering how and when to update $\overrightarrow{c_a}$, we must ask: is it necessary to update $\overrightarrow{c_a}$ at all? That is, after some time, do coordinates cease to change relative to one another, merely rotating about an axis, oscillating, or otherwise remaining stationary? The answer is no: coordinates do change, reflecting changes in the underlying network even over relatively short time-scales. We illustrate this change in Figure 9 by showing how four nodes' coordinates vary over time. The nodes are from four distinct regions. Their coordinates move in a consistent direction over a three hour period, neither rotating nor remaining within one area. Instead, this example portrays that $\overrightarrow{c_a}$ should be updated over time to sustain accuracy.

The fact that $\overrightarrow{c_a}$ must be updated suggests a trade-off between the drawback of changing $\overrightarrow{c_a}$, which induces (perhaps unnecessary) application-level work and $\overrightarrow{c_a}$'s accuracy. Our goal is to shift the line in Figure 7 to the left, increasing stability, without moving the line in Figure 6 to the right, increasing error.

Examining the correlation between triangle violations and stability suggests that coordinate movement, when it is not due to an underlying network change, is due to these violations. This makes sense because the violations mean that the coordinate cannot have an exact location. We found the average extent of a node's triangle violations correlate strongly with its average stability, measured in *ms* per update using an EWMA ($r^2 = .71$). This suggests that much coordinate change – but not all, as Figure 9 illustrates – is unnecessary and can be suppressed.

We examined four heuristics that each attempt to update $\overrightarrow{c_a}$ at appropriate times, dampening application updates while retaining the MP filter's low relative error. Two are based on simple thresholds and two on sliding windows of previous c_s coordinates.

5.1 Application Update Heuristics

We present four heuristics that each attempt to increase stability in application-level coordinates without decreasing their accuracy.

SYSTEM. If the change in $\overrightarrow{c_s}$ from one observation to the next is greater than a threshold τ, update $\overrightarrow{c_a}$. Thus, if $\|\overrightarrow{c_{s(t)}} - \overrightarrow{c_{s(t-1)}}\| > \tau$, let $\overrightarrow{c_a} = \overrightarrow{c_s}$. This heuristic is simple but suffers from a pathological case: many changes just under the threshold might occur, leading to high error.

APPLICATION. If the application's idea of the coordinate has strayed too far from the system's, notify the application. More precisely, if $\|\overrightarrow{c_a} - \overrightarrow{c_s}\| > \tau$, let $\overrightarrow{c_a} = \overrightarrow{c_s}$. This heuristic is a simple way to express that an update should occur when a drift in one direction occurs; it permits oscillations beneath τ.

RELATIVE. This is the first of the two window-based heuristics. RELATIVE measures the local relative distance compared with our nearest known neighbor r and updates the application if the change is larger than an error ϵ_r. RELATIVE averages each of its sets of coordinates by taking their centroid $\mathcal{C}(W)$. It computes, if

$$\frac{\|\mathcal{C}(W_s) - \mathcal{C}(W_c)\|}{\|\mathcal{C}(W_s) - \overrightarrow{r}\|} > \epsilon_r,$$

let $\overrightarrow{c_s} = \mathcal{C}(W_c)$. This heuristic exhibits three good properties: updates are relative to the node's locale, computing the centroid is inexpensive, and $\mathcal{C}(W_s)$ can be cached. The approximate nearest neighbor is learned through a comparison with each latency sample, where the node learns $\overrightarrow{x_j}$ and a new neighbor through gossip.

ENERGY. The last heuristic uses a statistical test that specifically measures the Euclidean distance between two multi-dimensional distributions [26]. It is based on the

energy distance $e(A, B)$ between two finite sets $A = \{\overrightarrow{a_1}, \ldots, \overrightarrow{a_{n_1}}\}, B = \{\overrightarrow{b_1}, \ldots, \overrightarrow{b_{n_2}}\}$:

$$e(A,B) = \frac{n_1 n_2}{n_1 + n_2} \left(\frac{2}{n_1 n_2} \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \|\overrightarrow{a_i} - \overrightarrow{b_j}\| - \frac{1}{n_1^2} \sum_{i=1}^{n_1} \sum_{j=1}^{n_1} \|\overrightarrow{a_i} - \overrightarrow{a_j}\| - \frac{1}{n_2^2} \sum_{i=1}^{n_2} \sum_{j=1}^{n_2} \|\overrightarrow{b_i} - \overrightarrow{b_j}\| \right)$$

Using this statistic, we can determine the divergence of the two windows. If $e(W_s, W_c) > \tau$, let $\overrightarrow{c_a} = \mathcal{C}(W_c)$. Computing this heuristic is more computationally intensive than RELATIVE, but the difference is negligible for the small windows we used.

5.2 Detecting Change with Windows

Ben-David, Gehrke, and Kifer propose an algorithm to detect when a stream of samples entering a database has changed [9]; their algorithm is similar to one proposed by Kleinberg for detecting word bursts in text streams [10]. The kernel of their idea is to divide a single data stream $S = \{s_0, s_1, \ldots, s_n\}$ into two sets (or windows), $W_s = \{s_0, \ldots, s_k\}$ and $W_c = \{s_{n-k}, \ldots, s_n\}$, that can be compared for statistically significant change using one of a handful of standard techniques (such as rank-sum).

The *start* window W_s holds the initial values seen and the *current* window W_c slides to include only the most recent values. By creating two distributions out of the single stream, a change in the underlying stream can be detected. Initially, both windows are empty. As each element s_i arrives, it is added to W_s and W_c until they are both of size k. When this size is reached, no more elements are added to W_s, and W_c slides to add s_i and drop s_{i-k-1}. With each new element, the sets are tested for difference. When the statistical test declares the two windows to be different, a *change point* is said to have occurred. At this point, both windows W_s and W_c are cleared and the process begins again. The tests Ben-David *et al.* examine in their work, however, are all for one-dimensional data. ENERGY and RELATIVE use tests for multi-dimensional data.

5.3 Summary of Application-Update Results

To examine how these four heuristics affected application stability and accuracy, we implemented them in our simulator and observed their behavior with different window size and threshold parameters. First, as expected, increasing the threshold required for application update increases stability but decreases accuracy. However, the window-based heuristics succeed in substantially increasing stability before any significant decline in accuracy begins. Second, large windows between 32 and 512 samples improve both stability and accuracy. Very large windows, however, cause too few updates to occur, decreasing accuracy. Third, the heuristics that do not use windows increase stability only at the expense of accuracy and are not robust to minor parameter changes.

5.4 Window-based Heuristics

Because the window-based heuristics, RELATIVE and ENERGY, are more complex, we examined their behavior first. We hypothesized that as the threshold for update increased, fewer updates of $\overrightarrow{c_a}$ would occur, leading to greater stability and perhaps reduced accuracy.

To examine how the thresholds τ and ϵ_r affect ENERGY and RELATIVE, respectively, we ran an experiment where we varied the value of the threshold and kept window size constant at 32. We recorded accuracy and stability; Figure 11 shows the median for both the distribution of median relative error per node and of instability. The results summarize the last two hours of the four hour trace.

The data establish that RELATIVE exhibits a near-linear increase in stability with increasing threshold. Thus, as RELATIVE requires more movement relative to the distance to the nearest neighbor, updates steadily decline. The increase in ENERGY's stability is curved but has no knee: it too exhibits a measured decline in coordinate change as the threshold to update increases. Both heuristics fall in the same range of relative error, with ENERGY exhibiting a more gradual decline as thresholds increase. Accuracy begins to decline for ENERGY after $\tau = 8$ and for RELATIVE after $\epsilon_r = 0.3$. These are the most conservative parameters that still grant an increase in stability, with 8% for RELATIVE and 34% for ENERGY.

Our second experiment with the window-based heuristics establishes boundaries for window size. Unlike the per-link MP filter, a large window is acceptable because windows grow with every observation, not with every link. However, very large windows are slow to react to true changes in underlying network conditions.

We ran an experiment in which we kept the threshold for application-update constant while we varied window size exponentially. We monitored accuracy and stability as before, and observed the frequency of application updates. This last number — that is, the number of times $\overrightarrow{c_a}$ is changed per unit time — is interesting because even though stability might be increased, it might not necessarily correlate with a decline in application notifications. Instead, stability could be increasing due to smaller updates occurring at the same frequency. We wanted to ensure that both instability and update frequency were decreasing because

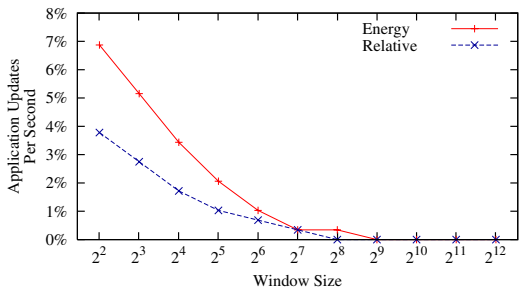

Figure 10. Varying window size

there is a cost for application notification. In Figure 10, we show the percent of the 269 nodes that changed their values $\overrightarrow{c_a}$ each second. The data show that not only do large windows ($\approx 2^5 - 2^9$) modestly improve accuracy, but they also result in a steady increase in stability and decline in update frequency. Across a wide range of window sizes, updates are both less frequent and cause less movement in aggregate, achieving two of the goals of the application-update heuristics. At a window size of 128 for example, RELATIVE's median relative error is 7% and its instability is $5ms/sec$, while causing only 1% of the nodes to be updated per second. This is a 42% increase in accuracy and a two orders-of-magnitude improvement in stability compared to the original algorithm. Because all large window sizes afforded a substantial improvement in the metrics, we chose the smallest of these, 32, to make a conservative comparison with the window-less heuristics and to use in our PlanetLab implementation. We used the threshold values gathered from the previous experiment.

5.5 Windowless Heuristics

The window-based heuristics have the disadvantage that they are slightly more complex than the windowless ones, SYSTEM and APPLICATION, and that they require more state. Using the parameters we established for window size from the previous experiment, we compared all four heuristics as we varied the update threshold and show the results in Figure 11.

Unlike ENERGY and RELATIVE the windowless heuristics could only directly trade off accuracy for stability and had a limited "sweet spot," that might change with a different trace. At low thresholds, when $\overrightarrow{c_a}$ is updated after only a small movement from its previous value, SYSTEM's and APPLICATION's performance remain similar to the raw MP filter. With a large threshold, $\overrightarrow{c_a}$ is rarely updated, leading to high error. Only at $\tau = 16$ do the two heuristics perform in the same range as the window-based ones. We conclude the added complexity and state of using one of the window-based heuristics is worthwhile because tipping in either direction results in poor performance on one of the metrics.

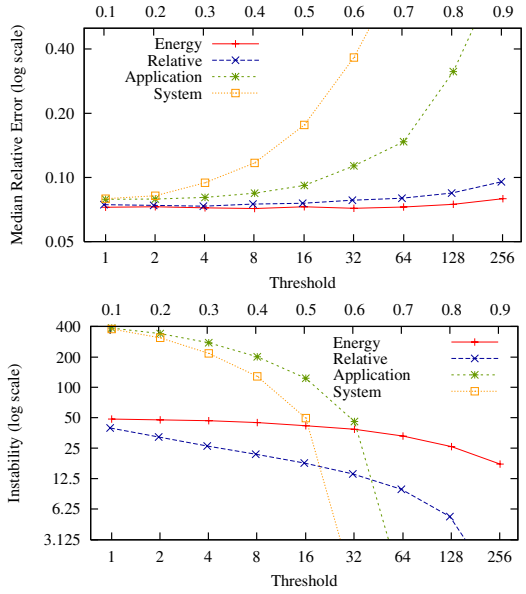

Figure 11. Varying threshold

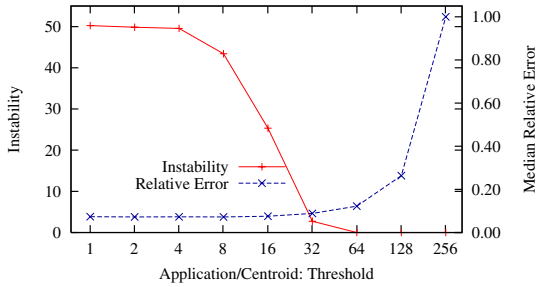

Figure 12. Application/Centroid

5.6 Discussion

Our primary goal in introducing the application-level heuristics was to further improve stability while maintaining accuracy. While we omit figures of traced-based simulation due to space constraints, the data from the PlanetLab experiment confirm that the two window-based heuristics achieve this goal (see Figure 13). Using the parameters established above, accuracy remains unchanged while RELATIVE and ENERGY shift the entire distribution of coordinate updates into a more stable regime.

Application-level accuracy and stability depend on both knowing when to update $\overrightarrow{c_a}$ and to what to set it. A substantial component of the success of the two window-based heuristics is their setting $\overrightarrow{c_a} = \mathcal{C}(W_c)$. One could argue that a simple threshold scheme might achieve similar performance if it too used the centroid of a collection of recent system-level coordinates. However, while it is true that all RELATIVE and ENERGY do is set $\overrightarrow{c_a}$ to the centroid of recent values for c_s, achieving the proper *rate* for these updates — knowing when to change — is a property simple thresholds

have difficultly performing.

To test this claim, we modified APPLICATION to set $\overrightarrow{c_a}$ to be the centroid of a window of the past 32 coordinates (the same size that ENERGY and RELATIVE use above). In our experiment, we varied the threshold at which updates were made and again monitored accuracy and stability. As the data in Figure 12 portray, this combined APPLICATION/CENTROID is more stable than APPLICATION and SYSTEM but, like the two window-less heuristics, it is not robust against slight changes in parameters and has high stability only at the expense of good accuracy.

6 PlanetLab Experiment

In order both to verify our simulator and to confirm that our findings were not limited to our latency trace, we implemented a version of NCs that run on a real network. This version uses application-level UDP pings as input, the same as our trace. Each node started with a small neighbor set and gossiped one address with every sample. Nodes sampled from their neighbor set in round-robin order at five second intervals. We added the MP filter and the ENERGY application-level update heuristic to our implementation. We used a window of 32 and $\tau = 8$ as suggested by the parameter space exploration in simulation.

In order to ensure a valid comparison between running NCs with our enhancements and without, we ran them on the same set of PlanetLab nodes at the same time, using different ports. One set of nodes used the MP filter and one did not; both used ENERGY. Because each node produced $\overrightarrow{c_s}$ and $\overrightarrow{c_a}$ with each sample, we could monitor the effects of the filter and the update heuristic separately. We ran this pair of coordinate systems for four hours on 270 PlanetLab nodes on June 24, 2005. We have subsequently been using the live coordinate system for significantly longer experiments on our work on streaming databases.

The results of the real-world experiment confirm those of our simulations. We show the relative error for the second half of the experiment in Figure 13. The data show that the MP filter reduces error and instability and the application-update heuristic further increases stability. With the MP filter, only 14% of nodes experienced a 95^{th} percentile relative error greater than one; without it, 62% did. ENERGY dampened the filter's updates: 91% of the time it fell below even the minimum instability of the raw filter. The enhancements combine to reduce the median of the 95^{th} percentile relative error by 54% and of instability by 96%. We also examined how latency and update filters affected these metrics over time; we show mean instability at ten minute intervals. The data show that after a half hour convergence period, the MP filter and ENERGY result in a much smoother and more accurate metric space on a real wide-area network. The data confirm that both enhancements have distinct effects on the two metrics and that both are required for a stable and accurate space from an application perspective.

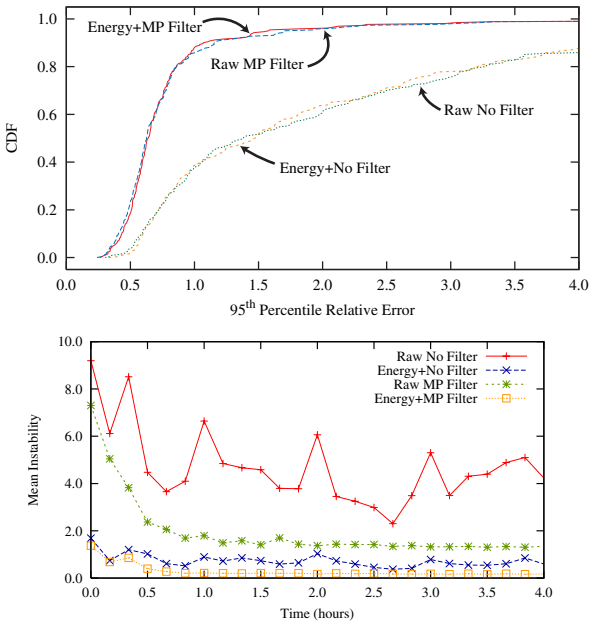

Figure 13. PlanetLab Stability and Accuracy

7 Related Work

Since Ng and Zhang provided the first examination of how to embed inter-node latencies in a metric space [15], a series of different approaches emerged. In their initial work, called Global Network Positioning, a coordinate space was built in two stages: first, a collection of well-known *landmarks* placed themselves in a vector space through all-pairs ping measurements; second, each joining node measured its distance to all of the landmarks and picked a coordinate that minimized the error to all of them. This approach does not allow for a smooth evolution of the space over time, nor is it decentralized. However, it did establish that, even with the error induced by triangle inequality violations, a high-quality space was possible. Lighthouses [18] Mithos [28], and NPS [16] extended the landmark approach by using multiple local coordinate systems, by building the space through preferring to measure nearby neighbors, and through a hierarchical architecture, respectively. More recently, Costa *et al.* developed PIC, another landmark scheme, which runs a Simplex solver on each node to minimize error [3]. PIC readjusts coordinates through periodically re-running this solver process and includes a test to defend its coordinate system against malicious participants. Cox *et al.* initially proposed Vivaldi [4] and Dabek *et al.* later improved its accuracy in two-dimensions with *height*, which was intended to explicitly capture the latency to a high speed link [5]. Shavitt and Tankel's Big-Bang Simula-

tions is an embedding technique similar to Vivaldi, although it models a potential force field instead of a mass-spring system [24]. Kleinberg has developed a theoretical grounding for network embeddings, analyzing how to embed coordinates with arbitrarily low errors [11].

Network embeddings were developed partially in response to the growing interest in topologically-efficient overlay routing. CAN's multi-dimensional space [22], in particular, motivated work on network-aware overlays and on using a node's network coordinates as its logical CAN coordinate [30, 31]. Other work has tried to solve similar routing problems without a coordinate space, arguing that maintenance is a burden or that error is high relative to a customized mechanism. In essence, this class of work solves the neighbor and routing problems *reactively*, through a spike in activity in response to an application-driven demand, while a long-running coordinate space solves them pro-actively. Meridian, for example, finds the nearest overlay node (*i.e.,* one running Meridian) to an arbitrary point in the Internet through a large set of pings in direct response to an application-level request [29].

In another effort to stabilize NCs, de Launois *et al.* modify Vivaldi to prevent oscillations in the presence of triangle inequalities [6]. They introduce a factor that asymptotically dampens the weight given to each new measurement, regardless of its source. While this factor does mitigate oscillations, it prevents the algorithm from adapting to changing network conditions as the pull of new measurements approaches zero.

8 Conclusion

In a real-world deployment, no fixed, single-valued latency matrix exists. Instead, nodes see a stream of latency values along each link. When these raw values are used to embed hosts into a metric space, the coordinate system they create is fragile.

Common techniques, such as excluding "large" values and using exponentially-weighted filters do not create a useful set of latencies. Instead, a short non-linear low-pass filter both removes extreme values and is agile enough to allow the output signal to accurately reflect changes in the underlying network. Additionally, the benefit of using more precise measurement tools is small relative to eliminating signal extrema with a low-pass filter.

We introduced update filters to manage triangle violations and examined the effect of four heuristics that determine how and when to update the application-level coordinate. The two heuristics, ENERGY and RELATIVE, that used a change-detection algorithm based on sliding windows best determined when to make the update. Additionally, using the centroid of a collection of recent coordinates set the application-level coordinate to a highly accurate value. We confirmed the results from our simulations with an implementation on PlanetLab.

References

[1] I. Abraham and D. Malkhi. Compact routing on euclidian metrics. In *PODC*, July 2004.
[2] Azureus BitTorrent Client.
[3] M. Costa, M. Castro, et al. PIC: Practical Internet Coordinates for Distance Estimation. In *ICDCS*, March 2004.
[4] R. Cox et al. Practical distributed network coordinates. In *HotNets*, November 2003.
[5] F. Dabek, R. Cox, et al. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, Aug. 2004.
[6] C. de Launois, S. Uhlig, and O. Bonaventure. A Stable and Distributed Network Coordinate System. Technical report, Universite Catholique de Louvain, December 2004.
[7] 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 Trans. Networking*, 9(5), 2001.
[8] S. Husen, R. Taylor, R. Smith, and H. Healser. Changes in geyser behavior. *Geology*, 32:537–540, 2004.
[9] D. Kifer, S. Ben-David, and J. Gehrke. Detecting Change in Data Streams. In *VLDB*, August 2004.
[10] J. Kleinberg. Bursty and hierarchical structure in streams. In *KDD*, July 2002.
[11] J. Kleinberg, A. Slivkins, et al. Triangulation and embedding using small sets of beacons. In *FOCS*, October 2004.
[12] J. Ledlie, P. Pietzuch, and M. Seltzer. Stable and accurate network coordinates. Research Report TR-17-05, Harvard University, July 2005.
[13] E. K. Lua, T. Griffin, et al. On the Accuracy of Embeddings for Internet Coordinate Systems. In *IMC*, Oct. 2005.
[14] A. W. Moore et al. Median Filtering for Removal of Low-Frequency Drift. *Analytic Chemistry*, 65:188, 1993.
[15] E. Ng et al. Predicting Internet Network Distances with Coordinate-based Approaches. In *INFOCOM*, June 2002.
[16] E. Ng and H. Zhang. A Network Positioning System for the Internet. In *USENIX*, Boston, MA, June 2004.
[17] L. Peterson et al. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets*, October 2002.
[18] M. Pias, J. Crowcroft, S. Wilbur, et al. Lighthouses for Scalable Distributed Location. In *IPTPS*, February 2003.
[19] P. Pietzuch et al. Network-Aware Overlays with Network Coordinates. In *IWDDS*, July 2006.
[20] P. Pietzuch, J. Ledlie, et al. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE*, April 2006.
[21] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting Network Coordinates on PlanetLab. In *WORLDS*, Dec. 2005.
[22] S. Ratnasamy, P. Francis, et al. A Scalable Content-Addressable Network. In *SIGCOMM*, August 2001.
[23] S. Rhea and D. Geels. Handling Churn in a DHT. In *USENIX*, June 2004.
[24] Y. Shavitt et al. Big-Bang Simulation for embedding network distances in Euclidean space. In *INFOCOM*, 2003.
[25] Y. Shavitt and T. Tankel. On the Curvature of the Internet and its usage for Overlay Construction and Distance Estimation. In *INFOCOM*, 2004.
[26] G. Szekely and M. Rizzo. Testing for Equal Distributions in High Dimension. *InterStat*, 5, November 2004.
[27] L. Tang and M. Crovella. Virtual landmarks for the internet. In *Internet Measurement Conference*, October 2003.
[28] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. In *HotNets*, October 2002.
[29] B. Wong et al. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, 2005.
[30] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-state. In *ICDCS 2003*, July 2002.
[31] B. Zhao, Y. Duan, L. Huang, et al. Brocade: Landmark routing on overlay networks. In *IPTPS*, March 2002.
[32] H. Zheng, E. K. Lua, M. Pias, and T. G. Griffin. Internet Routing Policies and Round-Trip-Times. In *PAM*, 2005.

This material is supported by the NSF under Grant No. 0330244.