A Locality-Aware Approach to Distributed Systems

A thesis presented

by

Jonathan Tormod Ledlie

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

September 2007

Thesis advisor
Margo Ilene Seltzer

Author
Jonathan Tormod Ledlie

A Locality-Aware Approach to Distributed Systems

Abstract

Today's high-bandwidth and real time applications place stringent, new demands on the Internet. For example, voice-over-Internet-Protocol, video-on-demand, content distribution, and real-time multiplayer gaming require real-time communication and dissemination across a potentially wide area. In all these applications, placing services, creating groups, or selecting servers in a fashion that takes network latency into account can dramatically improve performance.

Network coordinates are a promising technique for providing locality-awareness for these applications. They produce scalable latency estimates with minimal overhead. Previous work has shown the feasibility of network coordinates, but only in limited contexts. In this thesis, I measure the performance of the largest existing network coordinate system, improve its accuracy and stability through several key techniques, develop a locality-aware routing substrate, and build locality-aware applications with network coordinates.

I construct accurate coordinate systems in live networks. I introduce three techniques to improve the accuracy and stability of live coordinate systems and study their performance within Azureus, a popular BitTorrent client with more than a million nodes. Released as the open source Pyxida library, these techniques minimize overhead, adapt to latency anomalies, and increase coordinate stability, improving Azureus's accuracy by 43% and its stability by four orders-of-magnitude. Studying this system has also generated long-term traces for other researchers to use.

I also examine locality-aware routing and resource selection. I develop a practical routing algorithm using network coordinates, creating a building block for higher-level abstractions such as multicast and remote service discovery. I measure two applications that use network coordinates to optimize resource selection decisions: overlay routing, where delay was cut by 33%, and swarm-based file exchange, where network usage is reduced by 12% and download times are improved by 11% in my experiments.

Network coordinates are not always the best tool to provide locality-awareness. I conclude with an examination of the difficulties in directly embedding network characteristics other than latency — in particular, bandwidth — and qualify what contexts are appropriate for achieving locality-awareness with network coordinates.

Contents

List of Figures

List of Tables

Citations to Previous Publications

The bulk of Chapters 3, 4, and 5, and Section 6.2 appeared in the following three papers:

Stable and Accurate Network Coordinates, Jonathan Ledlie, Peter Pietzuch, and Margo Seltzer. In Procedings of the Twenty-sixth International Conference on Distributed Computing Systems (ICDCS), Lisbon, Portugal, July 2006.

Network Coordinates in the Wild, Jonathan Ledlie, Paul Gardner, and Margo Seltzer. In Proceedings of the Fourth USENIX Symposium on Network Systems Design and Implementation (NSDI), Cambridge, MA, April 2007.

Wired Geometric Routing, Jonathan Ledlie, Peter Pietzuch, Michael Mitzenmacher, and Margo Seltzer. In Proceedings of the Sixth International Workshop on Peer-to-Peer Systems (IPTPS), Bellevue, WA, February 2007.

Acknowledgments

Harvard has proven to be a fantastic place to dig in and go deep on a series of solid research projects — a place to "be the monk" as Remzi Arpaci-Dusseau, my advisor at Wisconsin, used to say. I have been able to explore what I wanted to at my own pace and I only hope that my future work will enjoy such intellectual freedom. Central to my experience at Harvard was Margo, who is the best advisor I could imagine. Her weekly "laser beam" of attention forced me to focus, but she never lost sight of me (or the rest of her students) personally.

I shared the Maxwell-Dworkin spaceship with an excellent crew. I have good memories of the early years with Jeff Shneidman, when we studied cryptography in the summer sun and then had the gall to whip together a paper telling our research community it was being eclipsed (again). My collaboration with Peter Pietzuch continues to be incredibly fruitful; I do hope he's enjoying his microwaved sausages in London. Other crew members I learned a great deal from include Kiran, David, Dan, and Rohan. Lex Stein never stopped being a source of entertainment. I'm looking forward to his first book.

Thanks to Paul Gardner from Azureus, Michael Parker from UCLA, and the folks who run PlanetLab for helping me study network coordinates "in the wild." Michael Mitzenmacher, Jim Waldo, Emin Gün Sirer, Neil Spring, Antony Rowstron, Miguel Castro and many anonymous reviewers greatly improved the quality of my work.

This thesis would not have been possible without the support I received from my friends and family. My college friends, Nick, Bill, Dave, Pove, and McKenna, were encouraging in their colorful, sardonic way. Thanks to the newer people in my Cambridge life — Jess, Rob, Nelly, Amanda, Claire, and Ben — and to the Southworth clan up North. Thanks to my brother Tim for carefully proofreading almost every word I have produced in the past six years and for putting up with all of my talk of nodes. Thanks to my parents for *everything* they have done. Karen met me at the beginning of my Harvard life: now we're married and have got a little person to boot. This especially would not have been possible without her support. Ever encouraging, I always kept her postcard on my desk, which said "You can do it!!!" And she was right.

Chapter 1

Introduction

In this thesis, I show how inter-node latencies in a live, large-scale network can be embedded into a coordinate space and how the resulting coordinates can be used to solve geometrically several common distributed systems problems. In particular, I show how to create a stable and accurate network coordinate system in large, real-world networks, characterize a live, million-node network coordinate system, and examine new methods for routing and resource discovery in distributed systems. I examine how to construct accurate live network coordinate systems and how they can assist distributed applications in making locality-aware decisions.

1.1 Motivation

Many contemporary distributed applications connect millions of individual machines into shared platforms, where part of the functionality exists on the endpoints — on users' machines — and the remainder runs *somewhere* between these endpoints, somewhere within the Internet "cloud." The choice of "where" within that "somewhere" can have enormous implications for application performance. For applications such as multi-player games, remote file backup and distribution, and video and audio streaming, naively treating the network as a black box results in poor performance for both the application provider and end users. Providers, such as content distribution networks (*e.g.,* Akamai [3]), typically must pay for the traffic their applications generate: positioning resources poorly in their network increases costs. Users also suffer when applications lack *locality-awareness* — knowledge of where users, data, and services exist within the network — through unexpected and unwarranted delays. Instead, applications should adhere to the *locality principle*, which states that network traffic with only local relevance should stay local [22], and purposefully consider where users, data, and services exist.

Consider locality-awareness in Voice-over-Internet-Protocol (VoIP). In VoIP, also known as broadband telephony, voice conversations are routed over the Internet, often via intermediary overlay services. VoIP users experience unreasonable delays if the bits that make up their conversations are routed circuitously. With two endpoints of a conversation whose positions are A and B, a goal of VoIP is to route traffic along the "line" between A and B. Deviations from this line will hurt performance metrics such as user-perceived delay. If no intermediate services are required, direct IP routing from A to B suffices. This will, in general, find a low latency path between the two and route around failures. However, if the VoIP application does have additional services, such as encryption, billing, or address book services, running *somewhere* in the space between A and B, it is important that traffic still adhere to this line and that services be positioned as close to it as possible. If there are hundreds or thousands of possible choices for these

infrastructure-providing servers, making good choices about which to use becomes a difficult challenge. Measuring all possible combinations of servers is not practical or scalable: for many applications, including VoIP, the choice of overlay providers must be made quickly, and the measurement cost for each choice must be low enough that many may be made concurrently. Network coordinates provide a solution to this problem: in this case, they can determine which servers are near the line and which are not. In general, they provide a quick and low overhead method for selecting among service providers in a large network in a locality-aware manner.

1.1.1 Network Coordinates: a Holistic Viewpoint

Network coordinates offer a new lens through which one can examine numerous distributed systems problems. Each network coordinate is a compact representation of a computer's position in a network. In a network coordinate system, a subset of inter-node latency measurements is embedded into a low-dimensional metric space. Each node maintains a coordinate, such that the metric distance between coordinates in the abstract space predicts real-world latencies.

Network coordinates offer a more holistic picture for depicting *sets of nodes* than other methods such as distributed hash table addresses (DHT) (*e.g.* a random number $\in (0 \ldots 2^{160}]$) and IP addresses. The power of network coordinates is their ability to imbue entire networks of nodes with a collective geometry that can be stored, calculated, and processed independently by every node. Thus, each node can make intelligent network-aware decisions using its local copy of this shared representation of the network.

Having local access to a geometric representation of a network allows nodes to perform standard distributed systems operations in a new way. For example, a node can now route *toward* a position that has a physical meaning, as opposed to an abstract IP or DHT address. In addition, one node can determine if pairs or groups of other nodes are (relatively) near one another. Non-participants may also be assigned coordinates so their location can play a role in application decisions such as where to place a service in a network. This local access to global locality information is a powerful new tool in distributed systems design.

1.2 Contributions

The contributions of this thesis are:

- A metric for describing the performance of network coordinate systems as they change over time. In particular, this metric describes the rate of coordinate movement, or *stability*, of an on-going system. I study the effect of two filters on stability using a PlanetLab environment consisting of several hundred participants.
- Design, implementation, and evaluations of methods for improving the accuracy and stability of live network coordinate systems. I measure how these methods affect a million-node network coordinate system and provide traces of this network for other researchers to use.
- An open source network coordinate library, called Pyxida. This library can be linked with applications that wish to use the methods I have developed. This library is now a part of the largest existing network coordinate system and several research projects.

- Adaptation of a routing algorithm designed for Euclidean spaces to live network coordinate systems, providing a basic routing primitive to higher-level mechanisms. I address imperfect network knowledge and churn and evaluate the algorithm on a realistic network.

- A study of two facets of using network coordinates for *anycast*. This study, together with an analysis of directly embedding and predicting bandwidth, forms the basis for a qualitative discussion of what types of applications make good targets for network coordinates and which do not.

1.3 Dissertation Overview

Network coordinate research, particularly research on live coordinate systems, is in its infancy. In Chapter 2, I cover the essentials of its brief history as necessary background. In particular, I focus on the mechanics of network coordinate construction, maintenance, and measurement.

The first half of the body of the thesis focuses on constructing accurate coordinate systems on live networks. An underlying goal of live network coordinate implementations is that each node must be able to develop its own stable and accurate approximation of its network, *i.e.,* an approximation that accurately predicts the location of participants, changing only when necessary. In particular, new methods are required to adapt to changes in the network while maintaining a stable set of coordinates. In Chapter 3, I propose a metric for network coordinate *stability*, or rate of change over time, and examine two new methods for stabilizing coordinates without reducing their prediction accuracy. The methods consist of two filters. The first filter, the *latency filter*, removes anomalous values from a stream of latency measurements between two hosts. This provides network coordinate algorithms with steady latency targets for each pair of neighbors. The second type of filter, the *update filter*, addresses a quandary common to applications that use network coordinates: given a change in the coordinate, should the application react or not? Is the change significant or is the coordinate moving but remaining in the same vicinity? The update filter is a layer that provides an answer to this question in the common case by exposing a stable, long-lived coordinate for application use. The central contribution of this chapter is to study the effect of these filters on prediction accuracy and stability using a live, continuously-running coordinate system on PlanetLab, consisting of several hundred participants.

Previous work on live coordinate systems left open the question of how they perform in truly large-scale environments [19, 46, 71]. By working with the developers of a million-node public BitTorrent network [15], I studied and refined a coordinate system four orders-of-magnitude larger than previous work. By tracing this system, called Azureus [6], I developed a more complete picture of how network coordinates perform "in the wild" and generated new data sets for other researchers to study. In Chapter 4, I analyze results from these data sets, compare my findings to previous models of coordinate behaviors, and propose several new techniques to address challenges encountered by live coordinate systems. The particular methods I propose minimize maintenance by using application traffic and address problems of churn, coordinate drift, intrinsic error, corruption, and latency variance. I implemented these methods as an open source library called Pyxida, which is now part of Azureus and research projects at University of California at Santa Barbara, Boston University, Microsoft Research Asia, and Singapore National University. The design, implementation, and evaluation of these methods and the characterization of Azureus' million-node network coordinate system constitute the main contributions of Chapter 4.

The second half of the thesis focuses on how network coordinates can assist applications through locality-aware routing and resource selection. Previous work suggested that network coordinates could be used as a basis for locality-aware routing: routing where the sender and destination have coordinates that, in

turn, depict locations in a network [2]. This is a powerful concept because many higher-level mechanisms, such as multicast, rely on a basic routing primitive. Chapter 5 starts with an existing algorithm designed for efficient routing in a two-dimensional Euclidean space [31]. I build on this theoretical routing work and develop a practical algorithm for routing using network coordinates, creating a building-block for higher level abstractions such as multicast and remote service discovery. The main contributions of Chapter 5 are generating a practical implementation of the routing algorithm that addresses imperfect network knowledge and churn and evaluating its performance on realistic networks.

While the previous chapters develop methods to improve network coordinate usability in terms of accuracy and routing functionality, Chapter 6 focuses on end-to-end application benefit. I highlight two applications where network coordinates assist in making resource selection locality-aware. The applications, DHT traversal and BitTorrent swarms, need to efficiently pick a "good" instance of a service from among many choices. The main contributions of Chapter 6 are measuring the extent to which network coordinates do help in these application-level decisions and a qualitative discussion of when network coordinates are appropriate and when they are not.

In Chapter 7, I discuss related work with a particular focus on coordinate evaluation metrics, coordinate-based routing, and other methods for making anycast decisions. In Chapter 8, I summarize my findings, draw conclusions from the results, and discuss future work.

Chapter 2

An Introduction to Network Coordinate Embeddings

A new class of distributed applications has emerged during the past decade. Examples of these new applications include voice-over-IP, video-on-demand, content streaming and distribution, remote file backup, and multi-player games.

These applications bring a new set of demands. Instead of forming client-server connections, they must create mesh-like network connections to neighbors. Some of these applications require that these neighbors be of low latency; for example, with VoIP, high latency connections result in jitter. For others, neighbors must be of high bandwidth; video downloads, for example, occur too slowly with insufficient bandwidth. With millions of concurrent users, the size of these applications is particularly new. These applications require scalable, efficient methods for finding other users to which they have low latency, high bandwidth, or both.

Research interest in network locality methods has expanded along with the growth of these applications. Before this growth, research on locality focused mainly on local area networks and shared-memory machines. In these contexts, co-locating data and processes was a difficult problem [10], but one that could be, to a great extent, holistically understood from a research perspective: all machines were under control of the same set of administrators and often in the same room! In contrast, the Internet is a shared, dynamic structure, so large that it is impossible to even approach the level of omniscience of local area environments.

Beginning in the mid- and late-1990s, researchers from different communities began trying to understand locality-awareness at the scale of the Internet. In particular, researchers from the Network Measurement and Theory communities studied the topology of the Internet — that is, the relative and absolute positions of objects — while researchers from the Systems community studied ways to improve application performance through locality-aware object placement. Network Measurement and Theory researchers examined the graph structure of Internet connectivity [4, 25, 43] and traffic patterns [34, 77], while Systems researchers examined the intelligent placement of Web caches [30], and developed content distribution networks [23], overlay networks [5, 36], and distributed hash tables [70, 74], which explicitly take aspects of locality and proximity into account. Over the past decade, research driven by these three communities has led to the emergence and development of network embeddings.

This chapter is organized as follows:

Section 2.1 reviews early work on latency prediction, which primarily attempted to make estimation scalable through clustering.

Section 2.2 discusses the first key work on assigning coordinates to nodes through a network embedding. This work showed the feasibility of network coordinates despite the existence of triangle inequality violations.

Section 2.3 covers the main developments in network coordinate research since this initial key work.

Section 2.4 provides an introduction to Vivaldi, which is the network coordinate algorithm used in the live coordinate systems I study in later chapters.

Section 2.5 reviews performance metrics for network coordinate systems.

Section 2.6 summarizes the particular parts of the background the reader should keep in mind for the rest of the thesis.

2.1 Latency Prediction

Currently, the main method for approaching locality-awareness at Internet-scale is through latency prediction. Performing all-to-all $O(n^2)$ measurements when n is on the order of thousands or millions is intractable. Instead researchers have designed methods to measure a characteristic, typically latency, to a small fraction of the overall network ($O(log\ n)$) and then base a locality prediction on these measurements.

While one might use *locality* to refer to physical location, inter-node bandwidth, or link loss, in the context of this thesis and in the context of previous research *locality* primarily refers to round trip time latency — that is, the time a small packet takes to reach a destination and return. Users, data, and services that have low latency to one another are local; those with high latency are distant. While *predicting* other network characteristics using other techniques remains a possibility, using coordinate embeddings to do so appears unlikely. I elaborate on the limitations of network coordinates in Section 6.4.

Early work on latency prediction focused on reducing the intractability of all-pairs measurements through clustering. Based on the assumption that nodes in the same cluster would have similar latencies to nodes in another cluster, researchers examined how to create clusters that accurately predicted latency and how to minimize inter- and intra-cluster measurement overhead. Francis *et al.* created clusters based on IP address prefixes, where representatives, or *landmarks*, from each prefix performed all-pairs measurements. They found, however, that prediction error was heavily dependent on the choice of representatives [27]. Chen *et al.* addressed this problem through the automatic formation of clusters and representatives; they found the cluster size and, more generally, the amenability of the network to clustering had the greatest effect on accuracy [14]. Ratnasamy *et al.* proposed a hybrid approach: nodes that are similar distances away from fixed landmarks place themselves into the same cluster; they also found error was highly dependent on the number of clusters [67]. Because all of this clustering involves measurement and lower network layers are already performing much of this measurement, Nakao *et al.* proposed reducing overhead by tapping into this existing information; unfortunately, this requires changes in the interface of Internet routers [54].

2.2 Embedding Latencies

These cluster-based approaches to latency prediction had major shortcomings: poor prediction accuracy, central points of failure, and variable results, depending on landmark selection. However, this work on landmark determination and clustering did lead to a new approach: turning physical inter-node latencies into abstract *network coordinates*.

Ng and Zhang provided the first examination of how to embed inter-node latencies in a metric space [57]. Their work, called Global Network Positioning, builds a coordinate space in two stages. First, a collection of well-known landmarks placed themselves in a vector space through all-pairs latency measurements. The l landmarks perform $O(l^2)$ round trip time measurements to one another. Then, offline, each landmark is assigned a coordinate such that it minimizes the (squared) relative error

$$\epsilon = \frac{| \, \|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij} \, |}{l_{ij}}$$

where i and j are landmarks, $\overrightarrow{x_i}$ and $\overrightarrow{x_j}$ are their coordinates, and l_{ij} is their round trip time. The error minimization is cast as a multi-dimensional global minimization problem and solved with a technique such as Simplex Downhill [56]. Second, each joining node measures its distance to the landmarks and picks a coordinate that minimized the error to them. Ng and Zhang's approach does not allow for a smooth evolution of the space over time, nor is it decentralized. However, their idea of assigning coordinates to participants by embedding their latencies into a metric space has remained at the core of the work in this area.

2.2.1 Triangle Inequality Violations

Unlike non-coordinate-based schemes for latency prediction, Ng and Zhang's research and other work on embedding latencies rely on limited triangle inequality violations in the latency distribution. The triangle inequality states that for any triangle the length of a given side must be less than the sum of the other two sides but greater than the difference of the two other sides, *i.e.,* the sides must be able to form a triangle. When the latencies between node triples cannot form a triangle, they are said to violate the triangle inequality. Because latencies in the Internet do not always obey the triangle inequality, any embedding of these latencies in a Euclidean space will necessarily exhibit some level of inaccuracy [81]. The Ng and Zhang work showed that, despite the natural and persistent existence of triangle inequality violations in the Internet's latency distribution, forming embeddings that predict latencies with low error is feasible, at least in simulation.

2.3 Approaches to Embedding Latencies

Since Ng and Zhang's work, a series of different approaches to embedding latencies have emerged. This work has split primarily over the use or absence of landmarks. In addition, there have been important developments both in evolving coordinates over time and in the geometry of the embedding space.

2.3.1 Landmarks

Using landmarks is simpler, easier to simulate, and at least as accurate as embeddings without landmarks; however, they introduce a central point of failure that is undesirable from an application viewpoint: they require tens or hundreds of well-distributed, stable participants, which has limited their appeal outside of the research community.

Several techniques have extended Ng and Zhang's landmark approach in important ways. Pias *et al.* developed the Lighthouse location mechanism, which determines network location without a fixed set of landmarks [62]. Instead, each node forms a local basis — its own coordinate system — using a random set of k other nodes, called lighthouses, and a transition matrix that converts from local bases to a global basis shared across all nodes. Pias *et al.* take a step toward eliminating landmarks. However,

they have high measurement and communication overhead because both the transition matrix and the $k \times k$ latency matrix must be kept up-to-date for each node. Ng and Zhang, the developers of GNP, refine their original work by making the architecture more scalable and the coordinates more stable over time [58]. Their solution is similar to that of PIC, which Costa *et al.* developed concurrently [16]. Both take a rigid stance on the correctness of a given node's coordinate: it is either computed (and correct) or not yet computed. In both Ng and Zhang and Costa *et al.* , once a node's coordinate has been computed, it can serve as a landmark for any other node. Ng and Zhang create a more scalable architecture by turning each node's dependence on its landmarks into a dependency graph; nodes only need to recompute their coordinates when a dependency recomputes its coordinate, increasing stability. They do not define stability, nor do they permit continuous coordinate refinement: nodes are not guaranteed access to a coordinate at all times. In addition to removing a fixed landmark infrastructure, Costa *et al.* focus on securing coordinate computation through the exclusion of malicious nodes. This problem of malicious coordinate corruption is still unsolved, as I discuss in Section 4.6.4.

2.3.2 Spring Relaxation

Non-landmark approaches have been integrated into several live applications, such as Azureus [6], the Chord distributed hash table [20], Stream-Based Overlays Networks [65], and the Bamboo distributed hash table [69], because they do not require a fixed infrastructure.

Concurrently, Shavitt and Tankel [71] and Cox and Dabek *et al.* [18, 19] developed network coordinate embedding methods that eliminate landmarks. Instead, both model the network as a set of massless bodies connected by springs, where the rest length of the spring for a node pair i, j is the latency $l(i, j)$. Through a stream of small adjustments, the coordinates of each node come to rest in a low energy, low error position. Both were inspired by Hoppe [32], who used spring relaxation to guide the reconstruction of three dimensional surfaces from sets of unorganized points. What differentiates the Cox and Dabek *et al.* work, called *Vivaldi*, is they show a low error embedding can be found even if almost all of the springs are removed. While Shavitt and Tankel's method is centralized, requiring $O(n^2)$ latency measurements, each node in Vivaldi can perform its $O(log\ n)$ measurements and adjust its coordinate independently. I cover the Vivaldi algorithm in greater depth in Section 2.4.

Much of the early coordinate embedding work depicts assigning coordinates as a one-time operation. Because the underlying latencies that they are meant to predict change over time, however, coordinates must be recomputed at some rate in order to remain accurate. Costa *et al.* discuss the need for coordinate renewal, but depict each renewal as a discrete system-wide event. In contrast, every nodes' coordinate in Vivaldi continuously and independently evolves, eliminating system-wide re-measuring, re-computing, and re-deployment, and periods of poor accuracy that would occur prior to each redeployment.

2.3.3 Coordinate Space Geometries

Most algorithms for network coordinates embed into a low dimensional Euclidean space. In a sense, the fact that Euclidean spaces yield accurate latency prediction is puzzling: the latency distribution is drawn from the Earth, a sphere. In practice, spherical coordinates are a poor choice, however, because the Internet's latency distribution does not conform to a sphere. For reasons more a result of politics and business decisions than geography, most packets travelling between Asia and Europe cross the Atlantic and the Pacific Oceans rather than go the more "direct" route. The fact that these paths cross through the U.S. makes the topology flat, rather than round (I examine the "flatness" of the Earth in more detail in Section 4.3.5). Because most long-distance traffic fans in and out of the U.S., researchers have tried hyperbolic coordinates

instead of Euclidean ones [72]: the U.S. acts as the hyperbolic space's "saddle." Because comparisons found neither a Euclidean nor a hyperbolic geometry dominates in all cases [50] and because Euclidean spaces are intuitively simpler, most research, including mine, uses Euclidean embeddings. Still, the reader should note that the choice of Vivaldi is orthogonal to the choice of the Euclidean geometry and that most of the techniques I propose in this thesis are tied to neither this specific algorithm nor geometry.

One useful alteration to the Euclidean distance function has been the addition of *height* [19]. With *height*, the distance between nodes is measured as their Euclidean distance plus a height "above" the Euclidean hypercube. While the main hypercube models the Internet cloud — that is, links *between* ISPs, height models the latency *to* the cloud, *e.g.,* DSL and cable links. With height, the distance between nodes i, j becomes:

$$\|\overrightarrow{x_i} - \overrightarrow{x_j}\| + h_i + h_j.$$

In empirical evaluations, height produces a significant boost in latency prediction accuracy.

Note that height does alter the geometry of the space, perhaps in a non-intuitive fashion. In particular, height should not be thought of as simply an additional dimension. Consider a system of coordinates that are two-dimensional plus height. The line from point a to b runs to the plane (where it is perpendicular to the plane), along the plane, and "up" to b: the line between two points never has a height of its own. Contrast this with three-dimensional coordinates, where the line between two points would be the direct, straight line between them. Intuitively, it is helpful to consider network coordinates as part of an n-dimensional geometry (without height), and that the addition of height describes the "price of admission" to the larger network where standard geometric notions make sense.

2.4 Vivaldi

Because Vivaldi is the network coordinate algorithm used in the live coordinate systems I study in later chapters, I provide a more in-depth description of its internals here.

Vivaldi models the network as a collection of springs that pull on each node's coordinate. Nodes adjust their coordinates through observations of their latencies to other nodes in the system, their neighbors. These observations can be explicit pings or may be gleaned from existing traffic. Through successive samples, each node refines its coordinate. Like a network of springs, coordinates become more accurate and stable with each successive adjustment. A network embedding with a minimum error is found as the low-energy state of the spring system.

The process of setting each coordinate appropriately has two main components: refining a coordinate with respect to a given neighbor and forming the neighbor connection graph.

2.4.1 Coordinate Refinement

In the process of gradually moving to the low energy spring state, each node refines its coordinate with respect to one neighbor at a time. Figure 2.1 illustrates the steps of node a refining its coordinate with respect to node b. The impact of this refinement is that the abstract coordinate space will better mirror physical reality at the end of the refinement than before it. As the figure portrays, the refinement goes through four main steps. The first two steps, measurement and reply, generate new information not previously available at node a: the current round trip time to node b and node b's current coordinate. The second two steps are internal to node a: based on this new information about node b, node a computes a *force vector* — a push or pull with respect to node b — and adjusts its coordinate accordingly. This process is repeated with each neighbor.

The heart of Vivaldi's coordinate refinement process is computing the force vector. Each node retains its coordinate $\overrightarrow{x_i}$ which is initially set to a random point close to the origin. All coordinates are the same low dimension, which is fixed *a priori*. In the coordinate refinement example in the figure, the abstract coordinates differed from physical reality: node a was $10ms$ too close to node b. Given that the coordinates are under constant adjustment and began at inaccurate positions, to what extent should the abstract coordinates be trusted to reflect the nodes' positions accurately? If a neighbors' coordinate is inaccurate with respect to the rest of the network, why should it be given much credence? A node's *error* w_i, or confidence, quantifies the steady refinement of its coordinate. Each node begins with an error of 1 and adjusts it to reflect the current prediction accuracy of its coordinate with respect to the rest of the known and measured network. As a node's coordinate more perfectly predicts latency, its *error* approaches 0.

The force vector is computed using a combination of a new physical round trip time measurement, l_{ij}, the pair of coordinates, $\overrightarrow{x_i}$ and $\overrightarrow{x_j}$, and the pair of confidences, w_i and w_j. The computation relies on an exponentially-weighted moving average (EWMA) to keep track of the running error with respect to the rest of the network. EWMAs capture a distribution's general trend by including all previous observations and giving them an exponentially-declining weight over time:

$$v_{t+1} = \alpha \times s + (1 - \alpha) \times v_t$$

where v_t is the current average and v_{t+1} is the value after including observation s. The statistic's behavior is controlled by one parameter, $0 < \alpha \leq 1$, which determines how much weight is given to the current observation.

Figure 2.2 details the steps through which these inputs yield a force vector from the perspective of node i, where node j is its neighbor. The algorithm consists of seven steps. 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, node i finds how far off the observation was from what was expected based on the pair of coordinates; this establishes the relative error ϵ of the coordinate pair (Line 2). Third, node i updates its confidence w_i with an EWMA (Lines 3-4). Unlike a typical EWMA, however, the weight α, is not fixed. Instead it is adjusted according to the trustworthiness of the coordinates, as captured by their confidence. Lines 5-6 compute the force vector. First, node i computes the pull on the coordinate, δ, based on i and j's confidence. Then, in line 6, δ dampens the magnitude of the change applied to the coordinate. The unit vector function u sets the direction to the vector pointing from node j to node i; the push or pull of the force is on this line. Constants c_e and c_c affect the maximum change an observation can exert on the confidence and coordinate, 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. Line 7 adjusts the coordinate itself.

2.4.2 Neighbor Connection Graph

Forming the neighbor connection graph is simple in theory, but can be quite tricky in live systems. The abstract model of Vivaldi is as a complete network of springs. To make the model scalable, however, Vivaldi removes all but $O(log\ n)$ springs per node: spring relaxation occurs with almost all of the springs removed. What remains is the neighbor connection graph. A random selection of neighbors, a random graph, performs well because this ensures a broad dissemination of information on the position of each node. In simulation studies of Vivaldi [19, 50], researchers typically formed neighbor connections omnisciently. In practice, forming a random subgraph would not be difficult — via linking to random destinations on a DHT

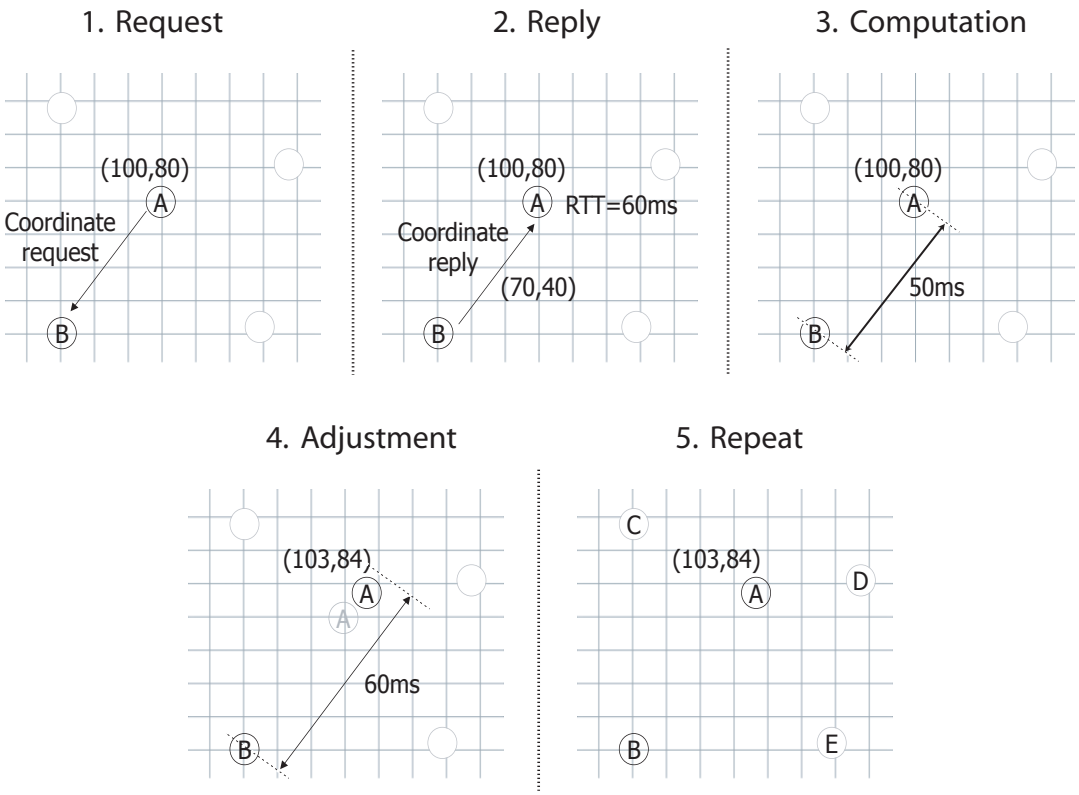

Figure 2.1: **Coordinate refinement process.**
Node a refines its coordinate with respect to node b. When nodes perform this refinement repeatedly, they steadily minimize global latency prediction error.

1. Request: Node a knows its own coordinate; it needs node b's coordinate and the current round trip time to b. Node a starts by asking b for its coordinate.
2. Reply: Node b replies with its own coordinate and, based on the time the exchange took, node a deduces the round trip time to b.
3. Computation: Now that node a has its own coordinate, b's coordinate, and a round trip time measurement, it can refine its coordinate with respect to b. First, it computes an estimate of the distances between them based on their coordinates. In this case, the coordinates say that the nodes should be $50ms$ apart. However, the measurement says they are $60ms$ apart. Based on this information, node a computes a force vector.
4. Adjustment: Node a shifts its coordinate away from b's coordinate using the force vector. This reduces a's prediction error to b so that next time a measurement is made to b, it will be more accurate.
5. Repeat: Node a refines its coordinate by repeating the same process with its other neighbors (c, d, and e).

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 $\overrightarrow{F} = \delta \times (\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}) \times u(\overrightarrow{x_i} - \overrightarrow{x_j})$
7 $\overrightarrow{x_i} = \overrightarrow{x_i} + \overrightarrow{F}$

Figure 2.2: *Vivaldi* force vector computation.

for example — but burdens the application with yet another set of nodes that require communication maintenance (*e.g.,* liveness heartbeats). Instead of these explicit pings to a neighbor set that is used exclusively for coordinate maintenance, it is preferable to reduce communication costs and use existing application traffic: coordinate update information can be "piggybacked" on to any request-reply application message that can approximate round trip time. Keeping a broad selection of neighbors while concurrently using existing application traffic is the focus of the *neighbor decay* technique I introduce in Section 4.4.1.

The end result of these two processes of coordinate refinement and neighbor establishment is that each node has a coordinate which can, to varying degrees of accuracy, predict the latency to any other node as long as that node's coordinate is available. Figure 2.3 shows a snapshot of a live embedding of 115 PlanetLab nodes that used Vivaldi to find their coordinates. By assigning coordinates to nodes in a network, applications approach a holistic viewpoint that aids in understanding the locations of components and where to place services in the network.

2.5 Measuring Coordinate Systems

Measuring live coordinate systems realistically can be surprisingly nuanced and dependent on context. Intuitively, the *accuracy* of a pair of coordinates is the difference between the predicted latency — the distance between the coordinates — and the actual latency. The error e for a pair of nodes i, j whose latency is l_{ij} is:

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

Depending on context, the accuracy for the system is the sum of these errors for all nodes, the sum of the error squared (the mean squared error), or the median for each node, among other statistics. Accuracy can also be normalized by dividing by l_{ij}; this *relative error* is ϵ in Figure 2.2. Unless otherwise noted, I will use relative error as the per-link accuracy metric because it facilitates comparison of a wide range of latencies. When looking at a distribution of relative errors for a whole system — especially when comparing across systems that might vary by a single parameter — I normally compare across a percentile or group of percentiles (*e.g.,* 50^{th}, 80^{th}, 95^{th}) in order to capture the broader shape of a particular change.

Attempting to compute an omniscient, system-wide "accuracy" fails in two respects on large, live systems. First, the latency l_{ij} is not a fixed quantity. As I discuss in Chapter 3, inter-node latency is not only time-dependent, but can also contain anomalies. Thus, any measurement of the accuracy of a particular pair

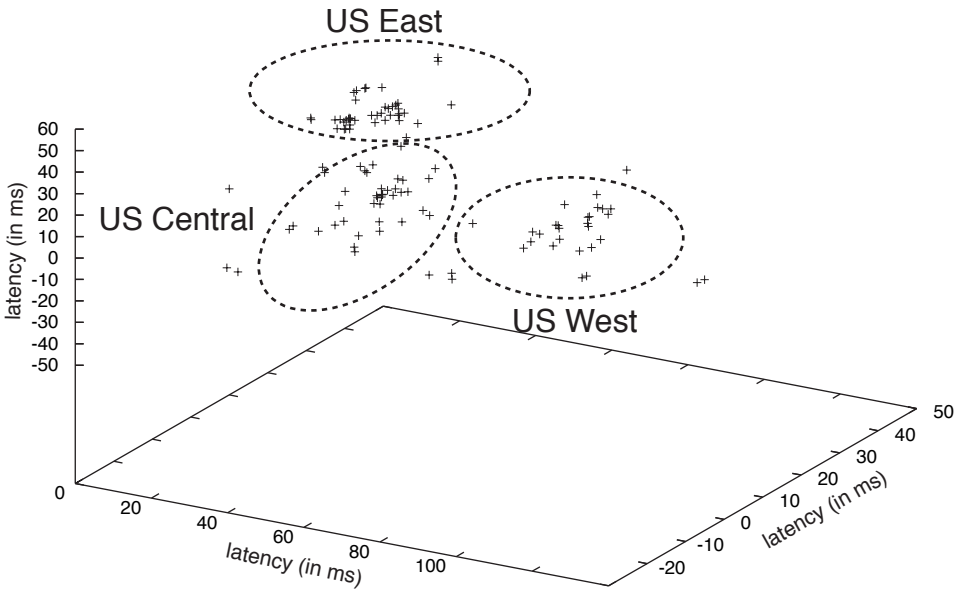

Figure 2.3: **Example of a three-dimensional Euclidean embedding.** Each point is a coordinate of a node in an embedding of 115 PlanetLab machines. The figure shows PlanetLab machines located in the United States; PlanetLab (as a whole) is a collection of approximately 500 machines spread around the world, located primarily at universities and research labs [61]. Note that the unit of every dimension is a time value; the distance between any pair of points is also a time: the latency prediction of the embedding. Because latency has a strong correlation with physical location and because PlanetLab machines in the U.S. roughly cluster into three groups, the coordinates of nodes in the embedding also form three clusters.

---

of nodes' coordinates must first find an acceptably accurate measure of their (current) latency. Second, in a large system, neither the inter-node latencies nor the coordinates for all nodes are known by any single observer. Instead, measurements of accuracy must aggregate many viewpoints from different observers of the overall system.

In simulation, the omniscient viewpoint is available. *Global relative error* is the accuracy from the viewpoint of an omniscient external viewer: at one instant, the metric is computed for all links. With the simulations that use a latency matrix, this is the metric that I use. On live systems, metrics, including relative error, can either be computed continuously — summarizing the distribution on-the-fly — or across a snapshot of the known network. While requiring less state, computing error continuously is problematic: (a) a single measurement may result in a large change in value and (b) it can become skewed by a handful of remote nodes if the "working set" of comparison is small. Instead of continuous error, I use *neighbor error* as a proxy for global error when live nodes are performing the computation themselves, *e.g.,* within the live Azureus clients in Chapter 4. Neighbor error is the distribution of relative errors for a set of recently contacted nodes.

I introduce the *stability* metric in Section 3.2 and review other metrics that have been applied to network coordinate systems in Chapter 7.

2.6 Summary

This chapter provided an introduction to latency prediction using network coordinates. In addition to summarizing the central pieces of research in this field, I gave an in-depth description of how the state-of-the-art network coordinate algorithm, Vivaldi, functions in live environments.

From this background, the reader should particularly keep in mind the following for the remainder of the thesis:

- The primary goal in latency prediction is to *efficiently* and *accurately* guess the latency between two endpoints in a network. "Efficiently" means that both the message overhead and state-per-node should be kept low, *i.e.,* messaging should occur on the order of seconds or minutes or "piggyback" on application traffic and state should be logarithmic in the size of the network. "Accurately" means that the error of guesses should be acceptably low for the given application. The error of a guess is the difference between the predicted latency and the actual latency.

- While much of the thesis focuses on practical systems-oriented issues with minimizing coordinate error, triangle inequality violations are a fundamental barrier to overall prediction accuracy. That is, if a perfectly accurate, static snapshot of latencies between all nodes in a network could be taken and the embedding performed offline, triangle inequality violations, which exist in all but the simplest networks, would prevent coordinates from providing perfect latency predictions.

- Vivaldi is a distributed network coordinate algorithm that runs in the same manner on all participants in a network coordinate system. It is most easily visualized as a spring relaxation process, where every node has a spring to every other and where the rest length of each spring is the latency between the endpoints. To make Vivaldi efficient, this spring relaxation occurs with all but $O(log\ n)$ of each node's springs removed. Each "spring" in Vivaldi is a connection to a neighbor. With each new measurement to each neighbor, Vivaldi computes a force vector, which is a small adjustment in the node's coordinate. After the force vector is added to the node's coordinate, the coordinate is more accurate with respect to the node's neighbors. Transitively, this also makes the node's coordinate more accurate with respect to the network as a whole. Many of the techniques and results in the thesis are independent of Vivaldi, but the reader should keep its internals in mind because it is the algorithm used in the live coordinate systems I study in later chapters.

Chapter 3

Accurate and Stable Coordinates on Live Networks

In this chapter, I introduce a new metric for measuring coordinate movement over time, *stability*, and describe how the addition of two types of filters produces coordinates that are stable, accurate, and adaptive to changing network conditions. These techniques yield high-quality network coordinates under live conditions.

The first filter, the *latency filter*, removes anomalous values from a stream of latency measurements between two hosts. This provides network coordinate algorithms with steady latency targets for each pair of neighbors. The second type of filter, the *update filter*, addresses a quandary common to applications that use network coordinates: given a change in the coordinate, should the application react or not? Is the change significant or is the coordinate moving but remaining in the same vicinity? The update filter is a layer that provides an answer to this question in the common case by exposing a stable, long-lived coordinate for application use. Applications can peek through this layer if they would benefit from more detail.

3.1 Introduction

In previous work, network coordinate schemes performed well only in simulation [57, 71, 76]. When run on live systems, however, the basic algorithms did not produce stable, accurate coordinates. The discrepancies between live and simulation performance were primarily a result of (a) the orders-of-magnitude variation in latency measurements between pairs of nodes that occur when running network coordinates 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 used derived latency matrices, typically containing the median values for links measured over hours or days. This previous work showed the feasibility of embeddings: if the right data could be fed in, good prediction would result. However, in live settings, poor input data was resulting in poor prediction: practitioners found themselves in a "garbage in, garbage out" situation. This chapter shows how to turn raw latency measurements into viable input for the embedding process in live settings. The chapter also describes and evaluates a layer for applications that wish to use network coordinates for prediction, while limiting their exposure to the underlying details.

This chapter is organized as follows:

Section 3.2 introduces a new metric for network coordinates. This metric, called *stability*, measures coor-

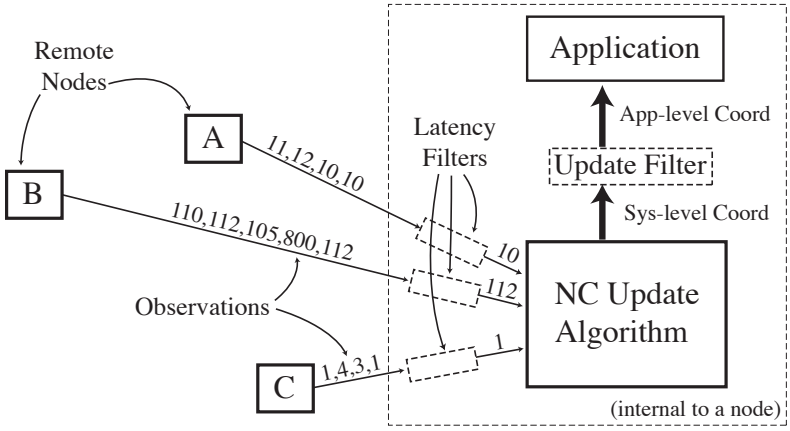

Figure 3.1: **Latency and Update filters.** Latency filters turn latency observations into useful network coordinates and Update filters notify applications only with significant coordinate changes.

dinate change over time.

Section 3.3 examines a latency distribution that exemplifies a typical input and discusses why network coordinate algorithms experience difficulty when used without a static latency matrix.

Section 3.4 presents a 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.

Section 3.5 differentiates between application- and system-level coordinates and compares four heuristics for improving application-level stability while maintaining accuracy. I 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.

Section 3.7 builds histories and application-level coordinates into an implementation that I run on a large network, resulting in a 54% improvement in accuracy and a 96% improvement coordinate stability.

Section 3.8 summarizes the results of this chapter.

For purposes of presentation and evaluation, I use *Vivaldi* as the canonical example of a network coordinate update algorithm (see Section 2.4 for an overview of Vivaldi). Because all embedding methods require latency estimates as inputs and produce coordinates as outputs, my techniques should be directly applicable to them as well. Figure 3.1 illustrates how the latency and update filters that this chapter discusses fit into a generic coordinate maintenance scheme.

3.2 A New Metric: Stability

Stable coordinates are particularly important when a coordinate change triggers application activity. In our distributed streaming query system, the *Stream Based Overlay Network (SBON)*, for example, a coordinate change could initiate a cascade of events, culminating in one or more heavyweight process

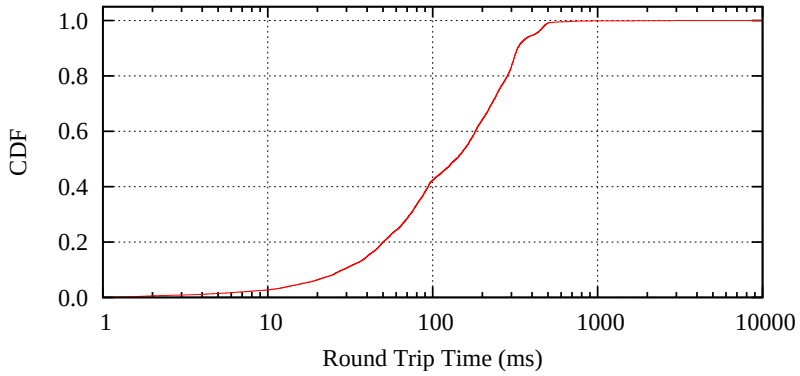

Figure 3.2: **Unfiltered latencies (all nodes).** The distribution of raw UDP latency measurements between PlanetLab nodes shows that a small fraction ($< 0.2\%$) of the measurements are greater than one second, which is longer than the common case even for inter-continental links. These measurements, when they occur irregularly, can poison the accuracy of live coordinate systems.

migrations [65]. If the systems' coordinates have not changed significantly, there is no reason to begin this process. 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 to quantify stability, s

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

where $\Delta \overrightarrow{x_i}$ is the metric distance from $\overrightarrow{x_i}$ at time τ to $\overrightarrow{x_i}$ at time $\tau + t$. The numerator is a measure of change in coordinate distance. It is the sum of the changes in distance for each coordinate during a time of duration t. The denominator is the time t, the duration of the measurement interval. Because the metric space is estimating latency, the distance between coordinates (the numerator) is in milliseconds. The denominator, the duration t, is also a time value. I denote stability in ms/sec unless otherwise noted.

3.3 Latency Measurements

When I first implemented live network coordinates, I found that lone samples, orders-of-magnitude greater than expected, periodically distorted the entire coordinate system. These instabilities appeared when raw latency data was fed into the update algorithm. Some, but not all, of the delay correlated with load spikes on the endpoints. The remainder appears to be due to standard in-network sources such as congestion.

Raw latency data show rare but persistent samples orders-of-magnitude larger than the common case. I collected a set of latency data from 269 PlanetLab [61] nodes over three days starting May 2, 2005, producing 43 million samples. To gather the trace, each node measured the latency to another node with an application-level UDP ping once per second.

I summarize the total distribution of measurements in Figure 3.2. The data show that a small fraction of 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 some links may be experiencing serious delays that network

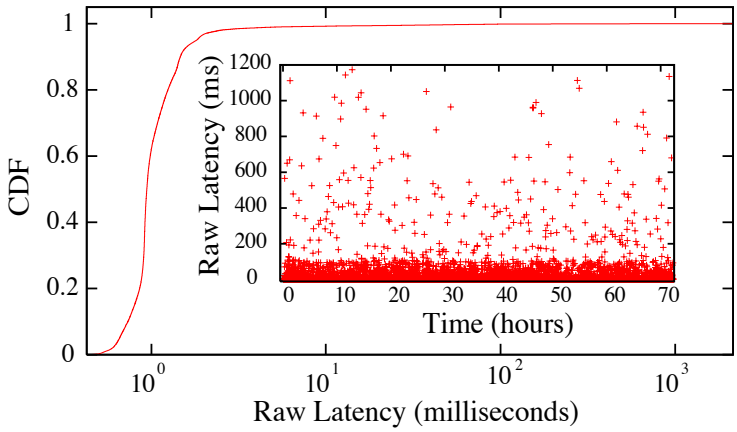

Figure 3.3: **Unfiltered latencies between a pair of nodes.** Some round trip time observations extend well beyond the median, even for nodes whose normal latency is very small. In addition, these infrequent order-of-magnitude delays are spread over time.

coordinates must automatically incorporate. The broad range of measurements severely curtails accuracy and stability.

I 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 a long tail as well. Figure 3.3 illustrates one representative link. It shows that some observations extend well 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 network coordinates should use as its approximation of the link latency, not the raw values. By giving network coordinates 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.

3.4 Filtering with Histories: Latency Filters

Based on my 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 [35, 53]. 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, I examined how the filter performed in term of prediction accuracy 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

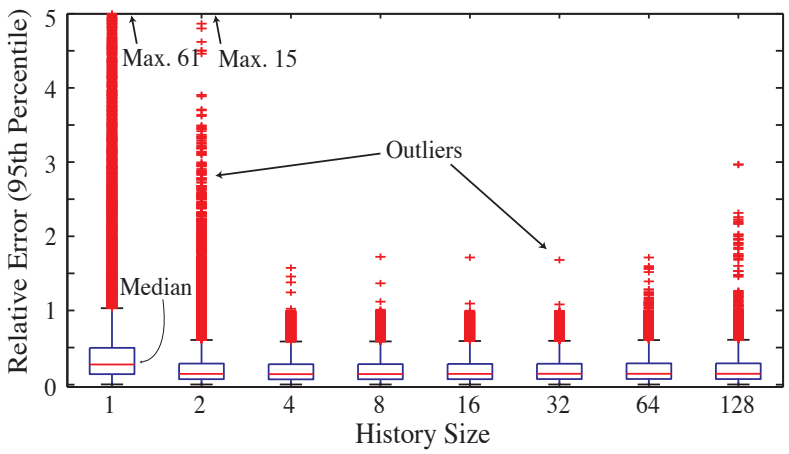

Figure 3.4: **MP filter prediction error.** Boxplots show the filters' ability to predict the following latency measurement for each link as history size h varies.

---

observation divided by the next observation.

I ran an experiment in which I varied the size of the window and the percentile used to predict the next value. Using the three day trace, I applied different filters to predict what the next observation would be and calculated the relative error between each prediction and the true observation. Figure 3.4 shows filters' abilities to predict the next latency for each link as I vary the history size h and keep the percentile $p = 25$. The results show that short histories, *e.g.,* only four observations, achieve the best performance (lowest error) with the fewest outliers. After performing a broad parameter search, I found 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 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, I 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 efficacy of the chosen p and h parameters over time. An adaptive solution would revisit the choice of p and h periodically to ensure that the filter remains a good predictor for future measurements.

3.4.1 Results from the Latency Filter

In order to compare network coordinates with and without the MP filter, I built a simulator that accepted the raw ping trace as input and mimicked the distributed behavior of Vivaldi. Through a comparison of running network coordinates on a real network and in the simulator, I 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

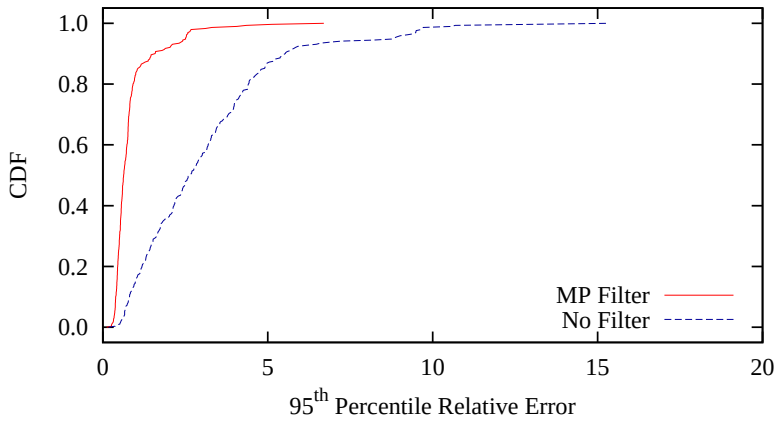

Figure 3.5: 95^{th} **Percentile Relative Error.** In a simulation that mimicked Vivaldi's distributed behavior over time, the MP filter greatly improved coordinate prediction accuracy. The figure shows a CDF of the instantaneous accuracy of each node's coordinate. Each data point is the 95^{th} percentile of the error of each measurement a node made during the second half of four hours of simulated time.

---

previous four observations (*i.e.,* the minimum) — I compared network coordinates with and without MP filtering. I ran network coordinates on a four hour section of the trace and show cumulative distributions for the second half of the run, eliminating start-up effects (I examine the rate of start-up in Section 3.7). I measured per-node accuracy and system-wide stability and summarize the results in Figures 3.5 and 3.6. 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 SBON, these distortions caused cascades of other updates to occur; the MP filter ameliorated this problem substantially. I show the impact of the MP filter on the overall latency distribution in Figure 3.7.

3.4.2 Other Filtering Methods

Before turning to the MP filter, I considered two methods that are commonly used to smooth out measurement error: thresholds and exponential averaging. Contrary to my initial expectations, these methods either had negligible impact on accuracy or stability or made conditions worse in some circumstances.

Thresholds. Prior to examining the latency distribution, I first considered using a fixed threshold to discard extreme values. Dropping all values above a threshold is simple and requires no state. Given the distribution of the entire trace (shown in Figure 3.2), this method can remove the most extreme outliers. However, each link tended to show its own set of outliers: most links exhibited long tails, but the shape of the distribution was different. For example, a cut-off of one second that might work for the general distribution would do little for outliers in the link shown in Figure 3.3, where the common case is less than $100ms$. Early in my study, I tried several thresholds before moving to more complex techniques.

EWMA. A commonly used statistic to smooth jittery data is the exponentially-weighted moving average (see Section 2.4). I added a per-link EWMA to the simulator with the goal that it would capture changes in network conditions while dampening the outliers. Table 3.1 shows the median value of the distribution of median relative error and stability when nodes use an EWMA filter with differing values of

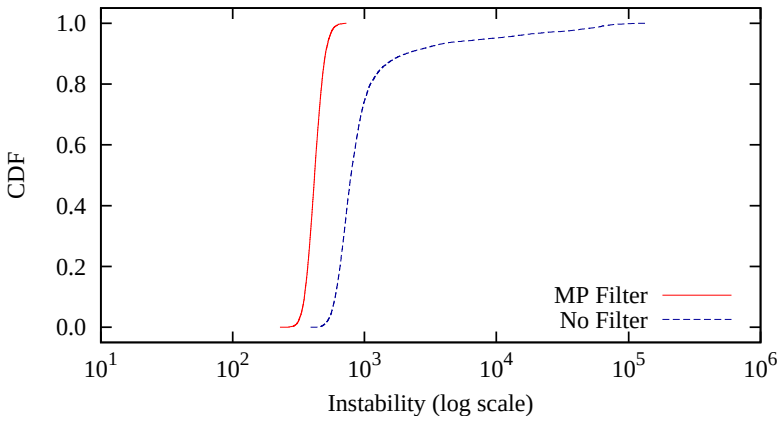

Figure 3.6: **Instability (log scale).** Again using the four hour trace, the MP filter reduced the long instability tail by three orders-of-magnitude, eliminating periodic distortions of the entire coordinate space.

Table 3.1: Filters: MP, Raw, and Exponentially-weighted

Filter	Median Relative Error	Instability
MP Filter	0.07 (−42%)	415 (−47%)
Raw (No Filter)	0.12 (0%)	783 (0%)
EWMA, $\alpha = 0.02$	0.27 (+125%)	490 (−37%)
EWMA, $\alpha = 0.10$	2.48 (+1960%)	1907 (+143%)
EWMA, $\alpha = 0.20$	5.70 (+4650%)	3783 (+383%)

α, as compared to using no filter and using the MP filter. The data show that even when an unconventionally low value for α is used, 0.02, smoothing with an EWMA still results in lower accuracy than using no filter at all. The outliers are not signifying a trend an EWMA should capture, but instead should simply be discarded.

3.5 Application-level Coordinates

Round trip time measurements that violate the triangle inequality have been shown to be a common occurrence on the Internet due to Internet routing policies. A recent study found around 18% of node triples of 399 PlanetLab nodes violate the triangle inequality [81]. Because of these violations, any network coordinate algorithm that refines its coordinate periodically, especially with every observation, will produce coordinates that oscillate in a region — decreasing stability — with the size of the region dependent on the size of the violation.

Using a latency filter greatly improved stability and accuracy for the set of network coordinates I examined. As Figure 3.6 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

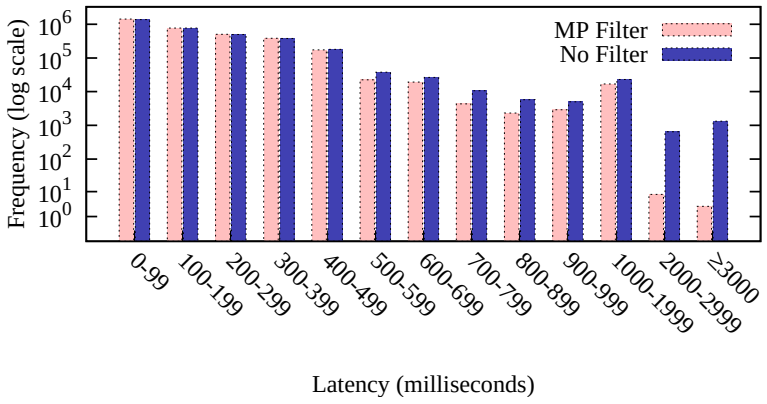

Figure 3.7: **Filtered vs. Raw Latency (ms).** When the MP filter is applied to the four hour latency trace, anomalous measurements are removed without an arbitrary threshold value.

---

the coordinate subsystem as a black box that only signals when there is significant change, one can limit application updates that, in turn, limit unnecessary application-level work. In my research group's 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 had 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, $\overrightarrow{c_a}$. Those in the former category would use $\overrightarrow{c_a}$ and the latter $\overrightarrow{c_s}$.

Before considering how and when to update $\overrightarrow{c_a}$, one 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. I illustrate this change in Figure 3.8 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 four 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. The goal is to shift the line in Figure 3.6 to the left, increasing stability, without moving the line in Figure 3.5 to the right, increasing error.

Examining the correlation between triangle inequality 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. I found the average extent of a node's triangle inequality violations correlate strongly with its average stability, measured in *ms* per update using an EWMA ($r^2 = .71$). I show this correlation in Figure 3.9. This suggests that much coordinate change — but not all, as Figure 3.8 illustrates — is unnecessary and can be suppressed.

I 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 sliding windows and two on simple thresholds. The sliding window heuristics compare windows (sets) of old coordinates to windows of new ones. The window of old coordinates is denoted W_s (start window) and that of new coordinates is

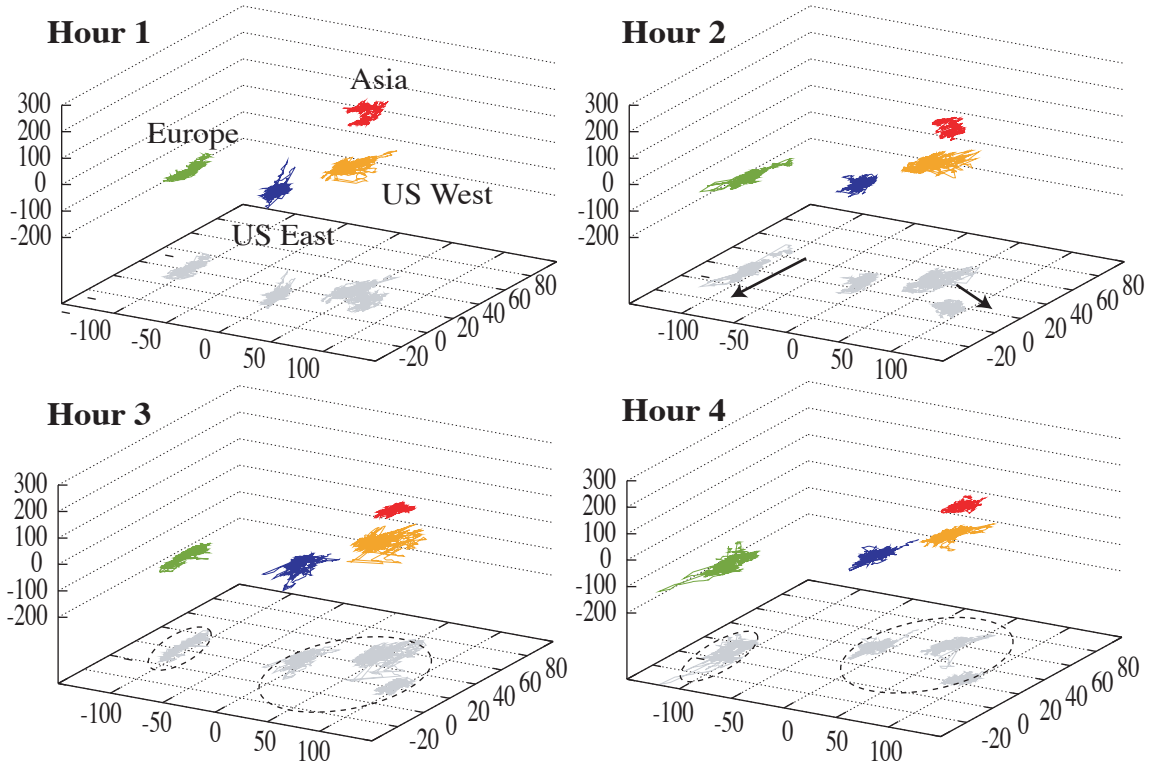

Figure 3.8: **Coordinate change over time.** Four coordinates from distinct regions move in consistent directions (marked with arrows) over a four hour period: Europe drifts away from the U.S. and Asia. For applications to maintain accurate coordinates, they must be periodically notified of underlying change.

denoted W_c (current window). These four heuristics each attempt to increase stability in application-level coordinates without decreasing their accuracy. I first present the sliding window heuristics; they performed best in practice. The two simpler heuristics are included second and used primarily for comparison.

3.5.1 Window Heuristic: Relative

This is the first of the two window-based heuristics. RELATIVE measures the local relative distance compared with the 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.

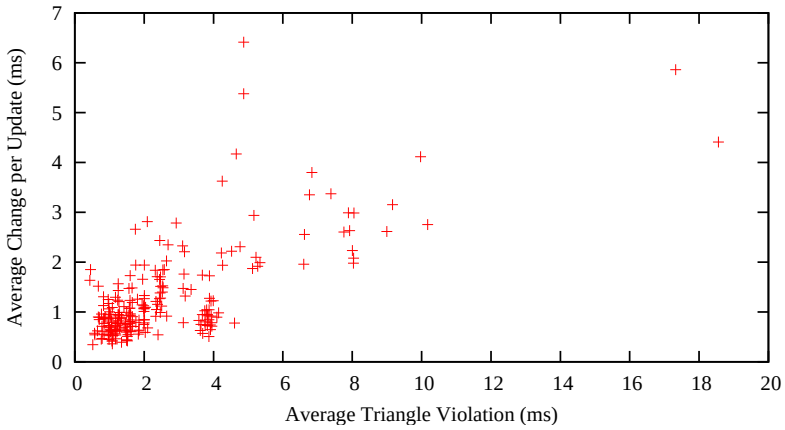

Figure 3.9: **Triangle inequality violations vs. Stability.** There exists a strong correlation ($r^2 = .71$) between the extent of a nodes' triangle inequality violations and their stability.

3.5.2 Window Heuristic: Energy

This heuristic uses a statistical test that specifically measures the Euclidean distance between two multi-dimensional distributions [75]. 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, one 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 used.

3.5.3 Threshold Heuristic: System

SYSTEM and APPLICATION are the two heuristics based on simple thresholds. 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.

3.5.4 Threshold Heuristic: 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.

3.5.5 Detecting Change with Windows

Ben-David, Gehrke, and Kifer propose an algorithm to detect when a stream of samples entering a database has changed [42]; their algorithm is similar to one proposed by Kleinberg for detecting word bursts in text streams [44]. 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 are for one-dimensional data; ENERGY and RELATIVE use a similar approach, testing for multi-dimensional data.

3.6 Application-Update Results

To examine how these four heuristics affected application stability and accuracy, I implemented them in the 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 can decrease accuracy. However, the window-based heuristics succeed in substantially increasing stability before any significant decline in accuracy begins. Second, windows between 32 and 512 samples improve both stability and accuracy because neither coordinate is in an unexpected location at the moment of comparison. 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.

3.6.1 Varying the Application-Notification Threshold

To examine how the thresholds τ and ϵ_r affect the four heuristics, I ran an experiment where I varied the value of the thresholds and kept window size constant at 32. I recorded accuracy and stability; Figure 3.10 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.

I hypothesized that as the threshold for update increased, fewer updates of $\overrightarrow{c_a}$ would occur, leading to greater stability and perhaps reduced accuracy. 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 similarly gradual: 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

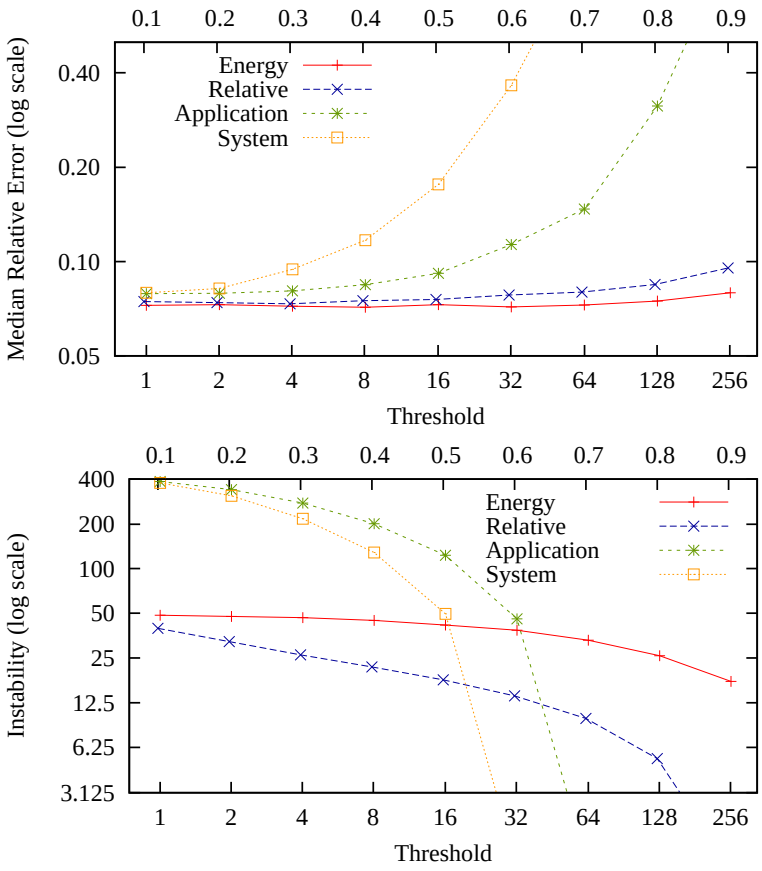

Figure 3.10: **Varying threshold.** ENERGY and RELATIVE, the two window-based heuristics, show significant improvements in stability before too-large windows begin to hurt accuracy. SYSTEM and APPLICATION can only directly trade-off accuracy for stability.

---

are the most conservative parameters that still grant an increase in stability, with 8% for RELATIVE and 34% for ENERGY.

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. 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. I 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.

3.6.2 Varying the Window Size

The 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.

I ran an experiment in which I kept the threshold for application-update constant while I varied window size exponentially. I monitored accuracy and stability as before, and also observed the frequency of application updates. This frequency — 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 shorter changes in distance occurring at the same frequency. I wanted to ensure that both instability and update frequency were decreasing because there is a cost for application notification. In Figure 3.11, I 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%, its stability is $5ms/sec$, while only 1% of the nodes are updated per second. This is a 42% increase in accuracy and a two orders-of-magnitude improvement in stability compared to the original situation. Because all large window sizes afforded a substantial improvement in the metrics, I chose the smallest of these, 32, to make a conservative comparison with the window-less heuristics and to use in my PlanetLab implementation. I used the threshold values gathered from the previous experiment.

3.6.3 Discussion

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 x$\overrightarrow{c_a}$ 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, I 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 the experiment, I varied the threshold at which updates were made and again monitored accuracy and stability. As the data in Figure 3.12 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.

3.7 PlanetLab Experiment

In order both to verify the simulator and to confirm that my findings were not limited to the latency trace, I implemented a version of network coordinates that run on a real network. This version uses application-level UDP pings as input, the same as the 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. I added the MP filter and the ENERGY application-level update heuristic to

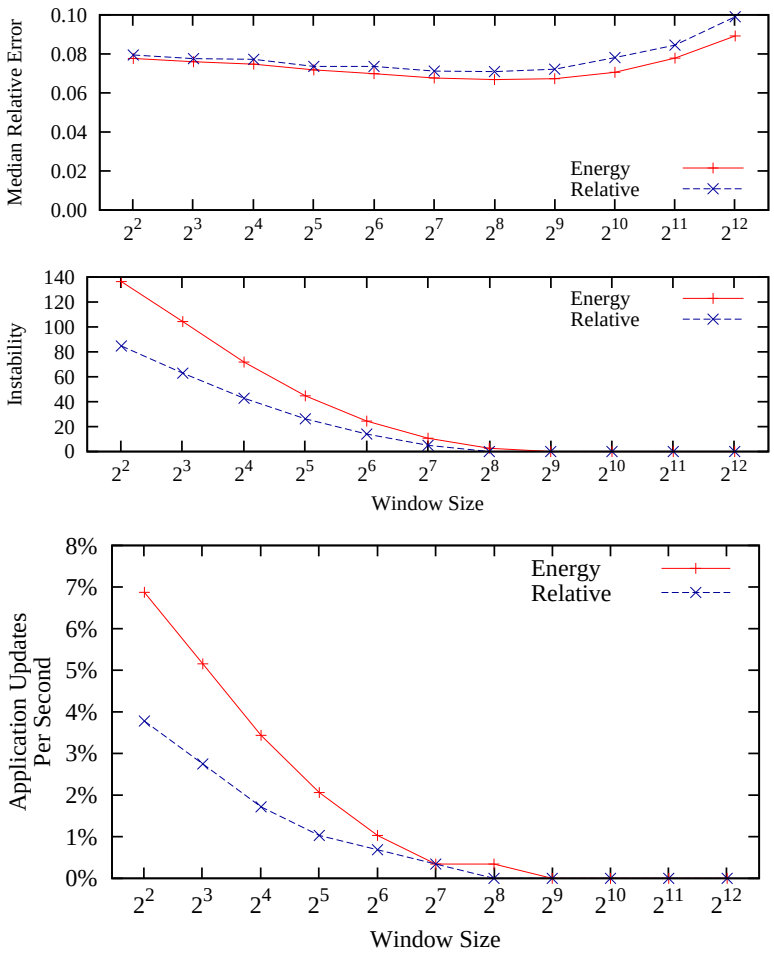

Figure 3.11: **Varying window size:** Median Relative Error, Instability and Application Updates per Second with varying window size for RELATIVE and ENERGY.

---

my implementation. I 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 network coordinates with my enhancements and without, I 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, I could monitor the effects of the filter and the update heuristic separately. I ran this pair of coordinate systems for four hours on 270 PlanetLab nodes on June 24, 2005. My research group and I 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 the simulations. I show the relative error and stability for the second half of the experiment in Figure 3.13. 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.

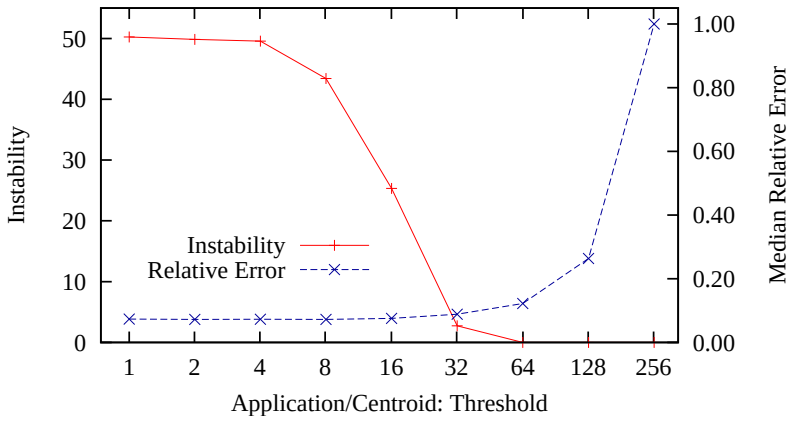

Figure 3.12: **Application/Centroid.** Without a good basis for when to perform an application update, APPLICATION/CENTROID achieves high stability only at the expense of good accuracy.

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%. I also examined how latency and update filters affected these metrics over time; I show median error and 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.

3.7.1 Discussion

After a close examination of all coordinate disruptions during the PlanetLab experiment, I discovered a source of much of the worst error. Most real-time low-pass filters add delay in order to incorporate future values. The MP filter outputted a value for every input, regardless of the history length: it produced the p^{th} percentile of the current state it was storing. Thus a pathological case occurs when an extreme outlier is the first observation for a particular link: even with the filter, this observation is what is used. In fact, this was the case for the five largest node displacements in the PlanetLab experiment and the echoes of these disruptions often continued for minutes. To compensate for this, network coordinates could wait until a sufficient number of samples are in the filter.

In simulation, I experimented with waiting until the second sample on a link to return an observation. This greatly reduced early instability, but, because the set of nodes was constant, it had only limited impact after start-up. In a long-running system where nodes periodically enter and leave, adding a delay to the filter would increase its robustness against these pathological cases. My implementations of latency filters in Azureus and Pyxida include this delay.

3.8 Summary

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

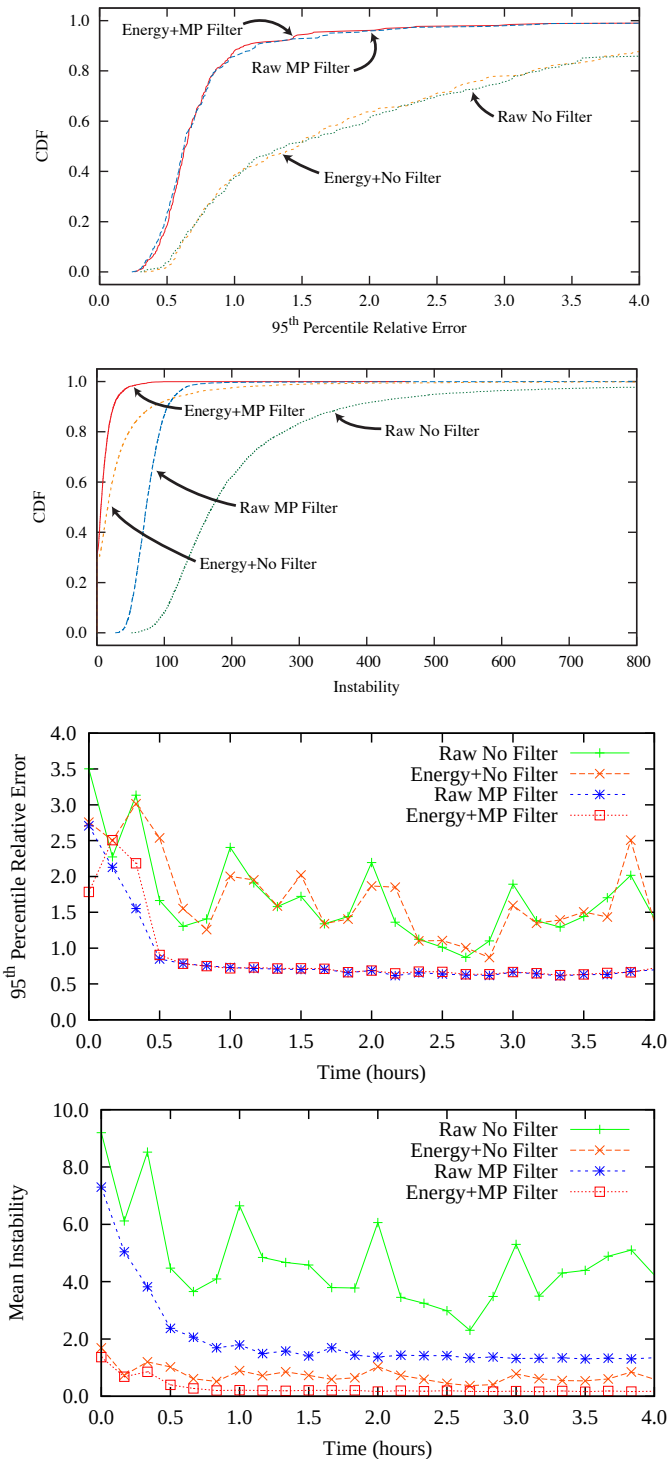

Figure 3.13: **PlanetLab Stability and Accuracy** The MP filter reduces error and instability and the application-update heuristic further increases stability. With the MP filter, only 14% of nodes experienced a 95^{th} percentile relative error greater than one; without it, 62% did. The enhancements combine to reduce the median of the 95^{th} percentile relative error by 54% and of instability by 96%.

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.

I introduced update filters to manage triangle inequality 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. I confirmed the results from simulations with an implementation on PlanetLab.

Chapter 4

Network Coordinates in the Wild

This chapter examines the performance of Internet-scale network embeddings through the study and refinement of a subset of a million-node live coordinate system. Many of the methods proposed in this chapter are now part of the Azureus BitTorrent network [6, 15], whose clients run the largest currently existing network coordinate system.

I conducted a long-term study of a subset of a million-plus node coordinate system and found that it exhibited some of the problems for which network coordinates are frequently criticized, for example, inaccuracy and fragility in the presence of violations of the triangle inequality. In Chapter 3, I introduced two new techniques, latency filters and update filters, and examined their performance on a small, live PlanetLab testbed. In this chapter, I introduce a third technique, neighbor decay, and show how these three methods combine to remedy many of these problems. Using the Azureus BitTorrent network as a testbed, I show that live, large-scale network coordinate systems behave differently than their tame PlanetLab and simulation-based counterparts. I find higher relative errors, more triangle inequality violations, and higher churn. I present and evaluate a number of techniques that, when applied to Azureus, efficiently produce accurate and stable network coordinates.

4.1 Introduction

Although network coordinates have attractive properties for latency prediction on the Internet, they have been criticized for requiring expensive maintenance and having prediction accuracy significantly worse than direct measurement methods such as Meridian [79]. At the very least, critics say that network coordinates are an unproven idea and unlikely to work in practice because Internet routing policies cause too many triangle inequality violations [81]. Supporters respond with claims that accuracies are reasonable ($8 - 15\%$), and they have demonstrated that coordinate maintenance can be built on top of existing application communication. They support these claims with simulations and small-scale live deployments on PlanetLab [19, 46, 61, 71].

This chapter provides the missing piece of the debate: data and analysis of a truly large-scale and long-running network coordinate system. The Azureus file-sharing network [6], which runs a million-node network coordinate system, is the main artifact for the analysis and experimentation. This work is the result of a collaboration between the Azureus team (Gardner) and a team from Harvard (Ledlie, Seltzer). Gardner contacted the Harvard team because Azureus was exhibiting some of the difficulties that we addressed in earlier work with a PlanetLab-based coordinate system [46] (included as Chapter 3). I merged the techniques from our previous work into the test branch of the Azureus code, used by approximately ten thousand clients.

While my previous techniques did work "in the wild," Azureus continued to experience unsatisfactorily high errors. This occurred because its gossip pattern stifled convergence: as all coordinate maintenance is "piggybacked" on other traffic, each coordinate became heavily skewed to small segments of the network and failed to become globally accurate. I created a new technique called *neighbor decay* that smoothly manages these skewed neighbor sets while retaining the appealing zero-maintenance property of Azureus' coordinates. With these techniques in place, Azureus' coordinates and, by inference, Internet-scale coordinate systems in general, can now tackle a basic goal: quickly and efficiently optimizing anycast decisions based on correct latency estimates. Because even with these approaches Internet-scale coordinates are still partially untamed, I isolated and analyzed a set of major remaining impediments.

This chapter is organized as follows:

Section 4.2 explains why practitioners, such as the Azureus developers, use network coordinates in large-scale deployments.

Section 4.3 uses a dense latency matrix to analyze the characteristics of the Azureus' latency distribution, determining its intrinsic dimensionality and the extent of its triangle inequality violations. This matrix provides a valuable new portal into Internet behavior. Previous large matrices were between DNS servers and did not capture latencies between actual nodes [19, 79]. I also provide evidence why Internet-scale latency estimation with coordinates works. I find the intrinsic dimensionality of large-scale systems to be less than previous work, which studied smaller networks [76], and I show why the world flattens into near-planar Euclidean coordinates.

Section 4.4 describes a new technique that came about through my observations of Azureus' live coordinate system. This technique manages neighbors and improves accuracy in coordinate systems where all gossip is "piggybacked" on existing traffic — *i.e.,* where there are zero maintenance messages and the where choice of neighbors is not under the control of the network coordinate algorithm.

Section 4.5 examines the live performance of Azureus through two methods: (a) Azureus clients I ran on PlanetLab and (b) crawling instrumented clients run by approximately ten thousand Azureus users. These improvements to the live Azureus coordinate system produced a 43% improvement in accuracy and a four order-of-magnitude improvement in stability.

Section 4.6 examines five primary causes of the remaining difference between the current live accuracy and what appears to be achievable based on simulation results. These barriers are churn, drift, intrinsic error, corruption, and latency variance. I present techniques for lowering these barriers and show how latency variance requires a fundamentally new approach to latency prediction.

Section 4.7 summarizes the results of this chapter.

4.2 Background: BitTorrent and Azureus

Azureus is currently one of the most popular clients for BitTorrent, a file sharing protocol [15]. For a given file, the protocol embodies four main roles: an *initial seeder*, *new seeders*, a *tracker*, and *peers*. Initial seeders, new seeders, and peers are all transient *clients*; trackers are typically web servers. The initial seeder is the source of the file. It divides the file into small pieces, creates a metadata description of the file and sends this description to the tracker. Peers discover this file description through some out-of-band mechanism (*e.g.,* a web page) and then begin looking for pieces of the file. Peers contact the tracker to

bootstrap their knowledge of other peers and seeds. The tracker returns a randomized subset of other peers and seeds. Initially, only the initial seeder has pieces, but soon peers are able to exchange missing pieces with each other, typically using a tit-for-tat scheme. Once a peer acquires all of the pieces for a file, it becomes a new seeder. This collection of clients actively sharing a file is called a *swarm*. In Azureus, file descriptors and other metadata are stored in a DHT, in which all clients participate, and any node can be assigned the role of tracker if it is or is near the root of the hash of a given file's descriptor. In practice, there can be many possible trackers from which to choose for a particular file and even more possible clients for a given piece.

The Azureus developers use network coordinates for two distinct purposes: (a) to optimize DHT traversal and (b) to select nearby nodes for application-level congestion monitoring. Future plans call for using network coordinates to optimize media streaming over Azureus and biasing the set of nodes the tracker returns to be nearby the caller, or *swarm localization*. In Chapter 6, I examine using network coordinates for DHT traversal and swarm localization.

I worked with the Azureus developers to analyze and improve the coordinates maintained by their system, which contains more than a million clients. I was able to modify the Azureus code internals and watch its behavior on a subset of the network because approximately ten thousand Azureus users run a plugin that automatically upgrades their version to the latest CVS release. According to the Azureus developers, the clients who use the latest release exhibit normal user characteristics, so I expect that my results generalize to the larger system.

Azureus uses the Vivaldi network coordinate update algorithm [19] (see Section 2.4 for a review of the Vivaldi algorithm).

4.3 Latencies in the Wild

Before I examine the accuracy with which Internet-scale latencies can be embedded into a coordinate space, I compare latencies in Azureus to those in other networks to gain insight into the causes of error in Internet-scale embeddings. I generated a dense latency matrix of a subset of Azureus and compared it to PlanetLab and to the MIT King data set, a square matrix containing the median latencies between 1740 DNS servers collected using the King method [19, 29]. Researchers found PlanetLab and MIT King can be reduced to low dimensional coordinates with $\leq 10\%$ median error [19, 46]. I examine three characteristics: inter-node round trip times, violations of the triangle inequality, and intrinsic dimensionality.

4.3.1 Collection

I instrumented Azureus clients that I ran on PlanetLab to record the application-level latency between them and the rest of the network, creating a dense latency matrix. These clients ran on 283 PlanetLab nodes for 24 days starting on July 19^{th} 2006, collecting 9.5×10^7 latency measurements to $156,658$ non-PlanetLab Azureus clients. To reduce these raw measurements into a dense latency matrix, I used the following process: first, I summarized each edge with the median round trip time for this edge, discarding edges with fewer than a minimum number of samples (4); second, I discarded all nodes that had fewer than half of the maximum number of edges (280). This process resulted in a 249×2902 matrix with 91% density, where 83% of the entries were the median of at least ten samples. I derived the PlanetLab data set from the Azureus matrix by simply selecting out its subset of hosts.

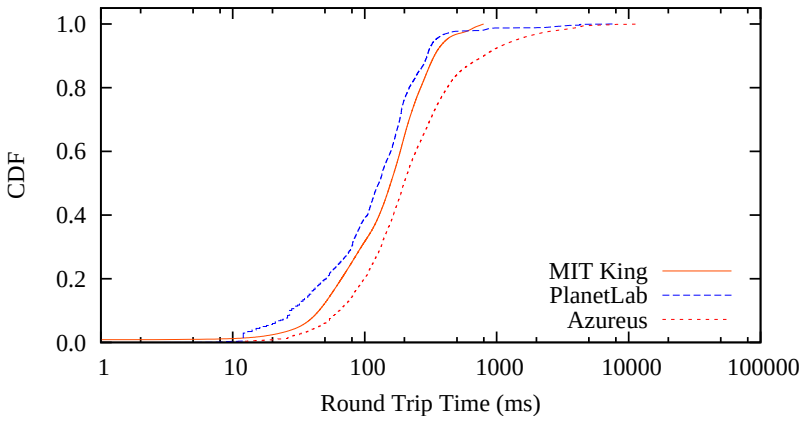

Figure 4.1: **Round Trip Time Comparison.** A comparison of round-trip times shows that Azureus spreads across a range one order-of-magnitude larger than MIT King, based on inter-DNS latencies. This larger spread tends to lead to lower accuracy embeddings.

4.3.2 Round Trip Times

In Figure 4.1, I illustrate the distribution of inter-node round trip times between nodes in the three data sets. The King measurements were limited to a maximum of $800ms$. The data exhibit one important characteristic: spread. The application-level, Azureus round trip times spread across four orders-of-magnitude, while the inter-DNS, King data set spreads across three. In theory, this is not a harbinger of higher embedding error; in practice, however, as Hong *et al.* have shown, the error between nodes whose distance is near the middle of the latency distribution tends to be the lowest [80]: with longer tails to this distribution, there are more edges to be inaccurate. (I found ICMP measurements exhibit a similarly wide distribution; see Section 4.6.5.) This wide spread is a warning sign that Azureus will have higher error than a system with a narrower round trip time distribution.

4.3.3 Violations of the Triangle Inequality

Network coordinate embeddings that use Euclidean distances make the assumption that the triangle inequality is not violated to a great extent by a large fraction of pairs of nodes (see Section 2.2.1 for an summary of the triangle inequality). Nodes with large and frequent violations tend to be the ones with the largest individual prediction error and their existence decreases overall accuracy.

I use a method from Tang and Crovella to examine the severity of triangle inequality violations [76]. This method normalizes the severity of each violation, permitting an all-pairs comparison. For each node pair, I find the shortest path between the two that passes through a third node. Thus, for all pairs of nodes i and j, I find the best alternative path through a node k and normalize by the latency between i and j:

$$rpl = min_k \left(\frac{d(i,k) + d(k,j)}{d(i,j)} \right)$$

Figure 4.2 illustrates the cumulative distribution of this quantity, the relative path length. Note that any fraction below 1 is a violation: there exists a path through an alternative node that is faster than the direct

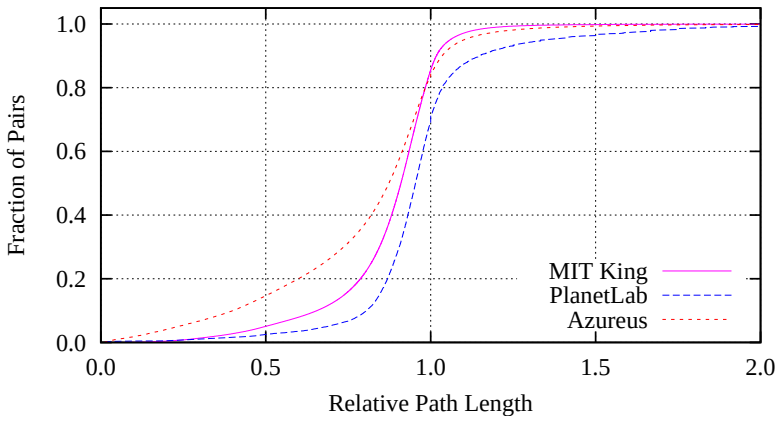

Figure 4.2: **Relative Path Length.** In all three data sets, over half of all node pairs fail the Tang/Crovella triangle inequality test, because there exists a third node between the nodes in the pair that produces a shorter path than the direct path between the two nodes. A large fraction of these violating pairs have paths that are significantly faster.

---

path. 83% of the Azureus pairs, 85% of MIT King, and 68% of the PlanetLab subset violate the triangle inequality. In contrast to earlier work that examined several small-scale data sets [76], I find the fraction of pairs with the largest violations to be quite large: Tang and Crovella found only 10% of nodes had an alternative path that is $\geq 20\%$ faster; here 37% of Azureus pairs and 22% of MIT King pairs exhibit this large level of violation.

I examined the cause of the large fraction of pairs with very low *rpl* (< 0.1) in Azureus. I found that only a few nodes were members of many of these low *rpl* pairs. What distinguished these nodes — and what was the cause of their frequent participation in triangle inequality violations — was that their delay to non-PlanetLab nodes was atypically large, on the order of seconds, while their delay to other PlanetLab nodes remained typical (less than a second). In effect, this extended one side of the triangles these nodes participated in: $d(i, j)$ became large while $d(i, k)$ and $d(k, j)$ did not. PlanetLab nodes that exhibited this behavior were co-located: Azureus traffic to non-PlanetLab sites was being artificially limited at these site gateways, while traffic to PlanetLab nodes avoided this traffic shaping. Rather than being a construct of the PlanetLab environment, this effect, leading to bi- or multi-modal latency distributions, will be the norm for at least some participants in Internet-scale applications that use well-known ports and consume a large amount of bandwidth, such as Azureus, because some sites will limit traffic and some will not. Like the round trip time spread, Azureus' violations foreshadow a higher embedding error.

4.3.4 Dimensionality

Network coordinates would be less useful if a large number of dimensions were needed to capture the inter-node latencies of the Internet. Tang and Crovella used Principal Component Analysis (PCA) to hint at the number of dimensions required to encompass this information for several small data sets [76]. Because I wanted to know if few dimensions would be sufficient for a large, broad spectrum of endpoints, I used the same method to examine the intrinsic dimensionality of Azureus.

PCA is a linear transformation from one coordinate system to a new, orthogonal coordinate sys-

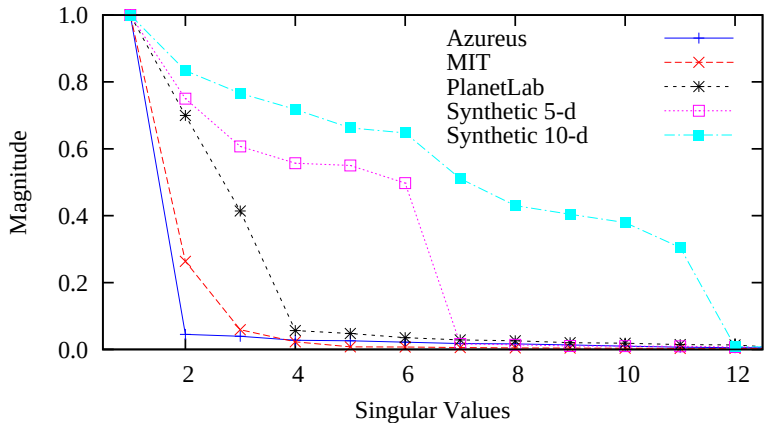

Figure 4.3: **Inherent Dimensionality.** Scree plots suggest the inherent dimensionality of MIT King, PlanetLab, and Azureus datasets is small. Two synthetic matrices of five and ten dimensions are included for comparison.

---

tem. The new system is chosen such that each subsequent axis captures the maximum possible remaining variance in projections from points in the old system to points in the new: the first new axis captures the most variance, the second less, and so on. While an input system of k elements will produce an output system also of k elements, often only the first several dimensions of the output system will summarize all or part of the same distance information of the original set of points. Singular values are a result of the PCA transformation: each new axis has a corresponding singular value that describes the amount of variance captured by this axis. Thus, if a singular value is very small or zero, this suggests that this axis is unnecessary in describing the variance in a particular data set.

Because PCA requires a full matrix, I first used the following two techniques to fill in the remaining 9% of the Azureus matrix and the missing 0.4% of the MIT matrix. I filled half of the missing Azureus values with the King technique [29] (King fails in certain cases, *e.g.,* when the endpoint cannot be resolved). I interpolated the remaining values in both matrices by embedding each matrix and extracting the missing values.

I use a scree plot to illustrate how much variance each new singular value is capturing, which in turn hints at the inherent dimensionality of the underlying data set. The independent variables of a scree plot are the singular values, sorted by their magnitude; the dependent variables are their corresponding magnitudes. At the point where the magnitude of the singular values becomes zero or nearly zero, the relative importance of this and subsequent singular values (*i.e.,* dimensions) is low. Up to this point, these dimensions are necessary to capture the values in the original input matrix, which in this case is made up of inter-node latency values.

I show the normalized singular values for the King, PlanetLab, and Azureus data sets in Figure 4.3. For comparison, I created synthetic $5d$ and $10d$ systems each containing 250 random points in a unit hypercube and found their singular values. As one would expect, the synthetic $5d$ and $10d$ data sets show a sharp knee soon after 5 and 10 singular values, respectively. In contrast, the bulk of the inter-node latency information from two Internet-based data sets requires very few dimensions. Azureus, in particular, is dominated by a single dimension, and MIT King by two. However, the next several dimensions remain significant for the few nodes that need to navigate around the clusters of nodes that have found accurate

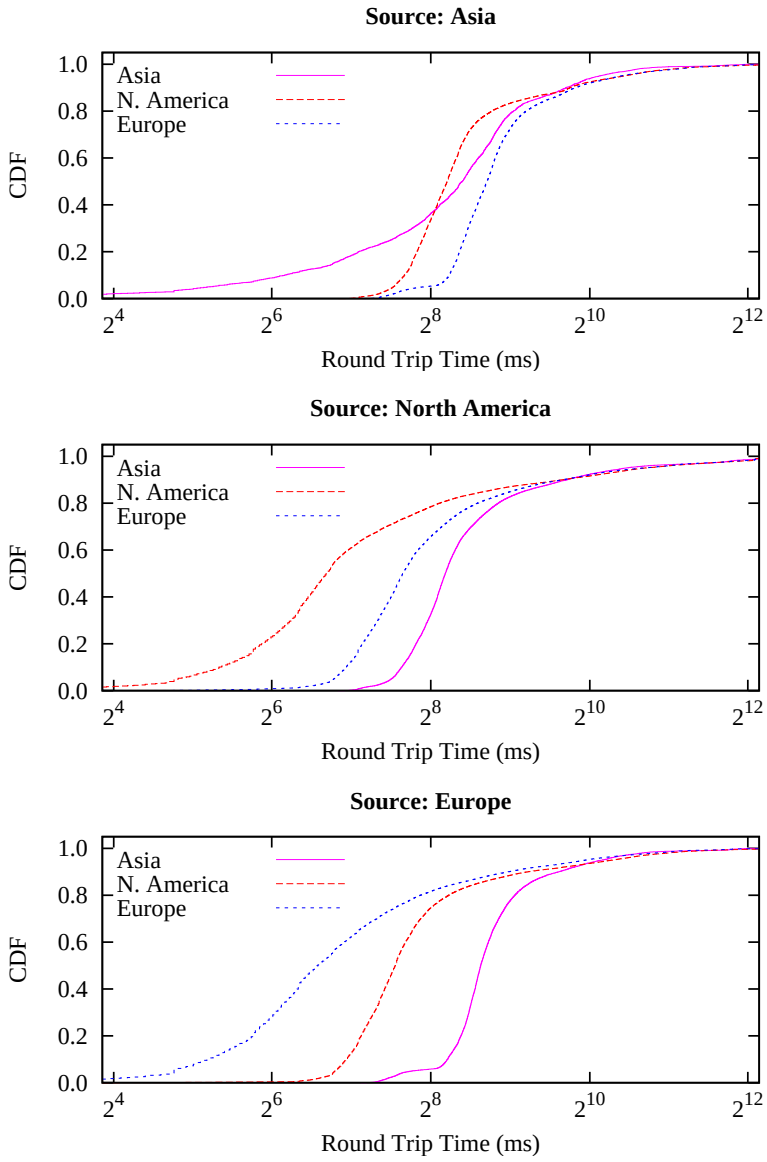

Figure 4.4: **Intercontinental Latency Distributions.** The plots illustrate why a Euclidean distance metric works for network coordinates on the Internet: messages from Asia to Europe (and from Europe to Asia) go through North America.

positions. In the data, this is shown by the continued relevance of singular values when compared to synthetic data sets. To lower the error for these nodes, I find $4-5$ dimensions is appropriate for Internet-scale network coordinates. While the previous two characteristics, round trip times and violations of the triangle inequality, suggest that the Azureus latency distribution will experience higher error than MIT King, its intrinsic dimensionality does not appear to be an additional impediment.

4.3.5 Intercontinental Latency Distributions

While the Azureus data set is clearly of low dimensionality, a more concrete way to examine the "flatness" of this large-scale network is to look at its intercontinental latency distribution. In a way, it is surprising that embedding latencies found on a globe (the Earth) into a Euclidean space works at all. If messages could be routed in any direction of the Earth's surface, using a Euclidean metric would be a poor choice. Previous work on spherical coordinates, however, found they had significantly larger error than Euclidean ones [19]. Anecdotal evidence suggested that the main reason why the Internet embeds into a low dimensional Euclidean space is because the world is flat: traffic between Asia and Europe flows through North America [19].

An examination of the Azureus data set confirms that this traffic flow is indeed the case. I mapped the IP addresses in the data set to countries through their autonomous system record and, in turn, mapped these countries to continents. As Figure 4.4 illustrates, *no* messages from Asia to Europe were faster than those from Asia to North America; the same holds in the other direction. All paths between Asia and Europe appear to travel in a line across two oceans. This trend continues until the speed of the connection to ISPs or other coarse delays begin to dominate.

This flatness suggests why hyperbolic coordinates [72] also work well: North America maps to the center of the hyperbolic space. Thus, because the distribution of latencies is "flat" – at least at a high level – using a Euclidean metric is sufficient. In the future, new direct transmission lines between Europe and Asia may change the Internet's shape, perhaps driving a shift to spherical coordinates.

Azureus' round trip time distribution and triangle inequality violations suggest that it will exhibit larger error than a DNS-based data set such as MIT King. However, I find that few dimensions are necessary to embed it with a Euclidean metric. Because its latency characteristics, in general, are similar to data sets that do embed with low error, there do not appear to be significant latency-driven barriers to forming an embedding with low error. Instead, I expect that application-driven barriers, such as churn, are likely to be significant contributors to inaccuracies in the coordinates' predictions. In the following sections, I examine the performance of Azureus' network coordinates both in simulation and in practice.

4.4 Taming Live Coordinate Systems

From my experience tuning the network coordinate system on PlanetLab in Chapter 3, I developed two techniques, latency and update filters, that led to more stable and accurate coordinates on a small "live" system [46]. The Azureus and Harvard teams worked together to integrate these techniques into the Azureus code. After confirming that these techniques worked as expected, I found and resolved a new problem: skewed neighbor sets. This problem particularly disrupts large-scale, live coordinate systems like Azureus that rely solely on other application communication for maintenance (*i.e.,* they have zero maintenance costs) and has has been suggested as a goal for coordinate systems [19]. Through experimentation with these techniques in simulation and periodic measurement of the live system, I arrived at coordinates that are not perfect, but are a satisfactory start. The reader may refer to the latency and update filters described in detail

in Chapter 3 for a review of these two techniques.

4.4.1 Neighbor Decay

Researchers have posited that a network coordinate subsystem could become a useful component of numerous large-scale distributed applications, particularly if it could perform its job *passively*, that is, without generating any extra traffic. In my Azureus implementation, this passivity was forced: I had no control over the selection of which nodes were gossipped with or when this gossip occurred, because the information necessary for a coordinate update was piggybacked on to other application-level messages, *e.g.,* DHT routing table maintenance. Due to this passivity and to churn, nodes did not have fixed sets of neighbors with which they could expect regular exchanges. In fact, nodes would frequently receive $1-3$ updates from a remote node as that node was being tested for entry into the routing table and then never hear from that node again. The net effect of these limited exchanges was that each node's "working set" was much smaller than the number of nodes with which it actually communicated. Nodes were having blips of communication with many nodes, but constant communication with few. The goal of *neighbor decay* is to expand the size of the working set, which in turn improves accuracy.

A standard, gossip-based coordinate update involves taking new information from a single remote node and optimizing the local coordinate with respect to that node. If some set of remote nodes is sampled at approximately the same frequency, a node's coordinate will become optimized with respect to these remote coordinates (which are in turn performing the same process with their neighbors). However, if some remote nodes are sampled at a far greater frequency than others, the local coordinate optimization process will become skewed toward these nodes. In the theoretical limit, the result would be the same, but in practice, these skewed updates – a problem that could be expected in any passive implementation – slow the global optimization process.

My solution to the problem of skewed neighbor updates is simple: the "springs" that are tugging on each node's coordinate gradually become less taut with age. Instead of refining the coordinate with respect to the remote node from which new information was just received, it is refined with respect to all nodes from which it has recently received an update. To normalize the sum of the forces of this *recent neighbor set*, I scale the force of each neighbor by its age: older information receives less weight. This allows nodes that are heard from only a few times to have a lasting, smooth effect on the coordinate. Algorithmically, I set the effect of a neighbor j on the aggregate force $\overrightarrow{F}$ to be:

$$\overrightarrow{F} = \overrightarrow{F} + \overrightarrow{F_j} \times \frac{a_{max} - a_j}{\sum(a_{max} - a)}$$

where a_j is the age of the knowledge of j and a_{max} is the age of the oldest neighbor.

This use of an expanded neighbor set that decays slowly over time has two main benefits. First, because the force from each update is effectively sliced up and distributed over time, nodes' coordinates do not jump to locations where they have high error with respect to other members of the neighbor set. Second, by keeping track of recent, but not old, neighbors, *neighbor decay* acts to increase the effective size of the neighbor set, which in turn leads to higher global accuracy. In my implementation, nodes expired from the *recent neighbor set* after 30 minutes.

Note the distinct effects of *neighbor decay* from both *latency* and *update* filters. Latency filters generate a current, expected round trip time to a remote node and update filters prevent system-level coordinate updates from spuriously affecting application behavior. Neighbor decay, in contrast, handles the problem of skewed updates that can occur when network coordinates are maintained as a passive subsystem. It allows the smooth incorporation of information from a wider range of neighbors, particularly in a

system where contact between nodes is highly transient. In simulation, I confirmed that neighbor decay substantially increased stability and moderately improved continuous relative error.

4.5 Internet-Scale Network Coordinates

A latency matrix tells only part of the story of an Internet coordinate system. It helps describe the network's characteristics, *e.g.,* its intrinsic dimensionality, but misses out on problems that may occur only in a running system, such as churn, changes in latencies over time, and measurement anomalies. I used two distinct methods to understand the online performance of Azureus' coordinates: (a) on PlanetLab, I ran instrumented Azureus clients that recorded the entirety of their coordinate-related behavior (Section 4.5.2) and (b) I crawled approximately ten thousand Azureus clients that internally tracked the performance of their coordinates using statistics I inserted into the Azureus code (Section 4.5.3).

4.5.1 Refining Azureus' Coordinates

Because updates to the CVS tree could take weeks to proliferate to a majority of users, changing single variables or techniques was not feasible. Instead I relied on simulations and on-going measurement to guide the roll-out of two major coordinate versions.

Azureus' coordinates originally used two dimensions, *height*, and none of the three filtering techniques I described in Sections 3.4, 3.5, and 4.4. I call this version 2D+H. To create version 5D, I incorporated the two techniques from my previous research, latency and update filters, into the code. Based on my on-going PlanetLab coordinate service, which did not use height and reliably exhibited low error, I also dropped *height* and added three more dimensions. Unfortunately, removing height proved to be a mistake. Through simulations of the Azureus latency matrix (see Figure 4.5), I realized I could expect a substantial improvement in accuracy by converting the last dimension of the $5d$ implementation to height without changing the gossip packet structure. I also found the highly skewed neighbor sets slowed convergence and developed the *neighbor decay* technique to compensate. I combined these changes and rolled out version 4D+H. The timeline for the major changes and experiments is in Table 4.1.

4.5.2 PlanetLab Snapshots

I took snapshots of each version by running clients on approximately 220 PlanetLab nodes. Each snapshot lasted for at least three days, and logged updates with approximately $10,000$ Azureus nodes. I collected a snapshot for each of the three versions in March, July, and September 2006, respectively. Note that these instrumented clients never stored or transferred any content that travels over the Azureus network.

I compare data gathered from the different versions in Figure 4.6. Because the data are aggregated across roughly the same source PlanetLab nodes, the three snapshots provide a reasonable, though imperfect, way to isolate the effects of the different techniques. In all cases, I find 4D+H is more accurate and stable than both the original 2D+H and the initial rollout of 5D.

My first revision, 5D, had mixed results. Based on this data and on simulations with and without height (see Figure 4.5), the data convey that the removal of height damaged accuracy more than the filters aided it. In retrospect, given the Azureus round trip time distribution (see Section 4.3.2), in which 7.6% of the node pairs exhibit round trip times ≥ 1 second, it is not surprising that using height helped many nodes find a low error coordinate. In addition, given that two dimensions are enough to capture much of Azureus' inherent dimensionality, it is also not surprising that the addition of three dimensions did not

Table 4.1: Timeline of Study and Refinement of Azureus' Coordinate System

December 2005	2D+H coordinates in place; inaccurate and unstable
March 2006	Preliminary data collection and simulation
May 2006	Released 5D coordinates Added Latency and Update Filters. Removed height. Probe-able, on-board statistics. Run in parallel with 2D+H.
July 2006	Long-term data collection Collection PlanetLab-to-non-PlanetLab matrix; probed on-board statistics.
August 2006	Released 4D+H coordinates (replace 5D, keep 2D+H. Added new technique: neighbor decay. Re-introduced height (both tested offline using matrix). Probed on-board statistics and ran clients on PlanetLab for detailed view.
December 2006	4D+H Coordinates become part of main-line Azureus code. Used concurrently by more than one million end hosts.
January 2007	First test of application-level performance: DHT Traversal.
May 2007	Second test of application-level performance: Swarm Localization.

radically improve accuracy. Although the 5D coordinates are less accurate, they are more than $2\frac{1}{2}$ orders-of-magnitude more stable because the latency filters prevent anomalous measurements from reaching the update algorithm.

My second change was more successful. The introduction of *neighbor decay* and the re-introduction of height in 4D+H create a much more accurate coordinate space than either of the previous two snapshots. This increase in accuracy occurs because neighbor decay enables nodes to triangulate their coordinates with a larger fraction of the network (and their neighbors are doing the same) and because *height* supplies the numerous nodes on DSL and cable lines with the additional abstract distance over which all their physical communication must travel.

I first evaluated *neighbor decay* in simulation. To confirm its continued effectiveness in a live system, I performed an experiment where I monitored the convergence of a node with and without *neighbor decay* enabled as part of the 4D+H coordinate system. In an average of three trials, I found neighbor decay improved median accuracy by 35%, 40% and 54% at the 15, 30, and 60 minute marks respectively.

4.5.3 End-Host Live Coordinates

The logs from the Azureus clients running on PlanetLab nodes provide a detailed view of a narrow slice of the system. To obtain a picture of the broader system, I inserted online statistics collection into the Azureus CVS tree. Using its recent neighbor set, each node computed its neighbor error and stability statistics on demand when probed. I present results from Azureus end-hosts running version 4D+H.

Figure 4.7 "live (all)" illustrates the data from a crawl of 9477 end-hosts. I exclude live nodes with fewer than 10% of the maximum 512 neighbors because their metrics are skewed to a very small percentage of the network. The data show that the bulk of the Azureus system experiences accuracy similar to clients running on PlanetLab. However, the error on the greater Azureus network has a long tail: at the 95^{th} percentile, its accuracy is 76% worse. As I discuss in Section 4.6.1, I conjecture that the high rate of churn causes much of this difference in the tail.

In order to hint at the exigencies caused by running "in the wild" as opposed to safely in the

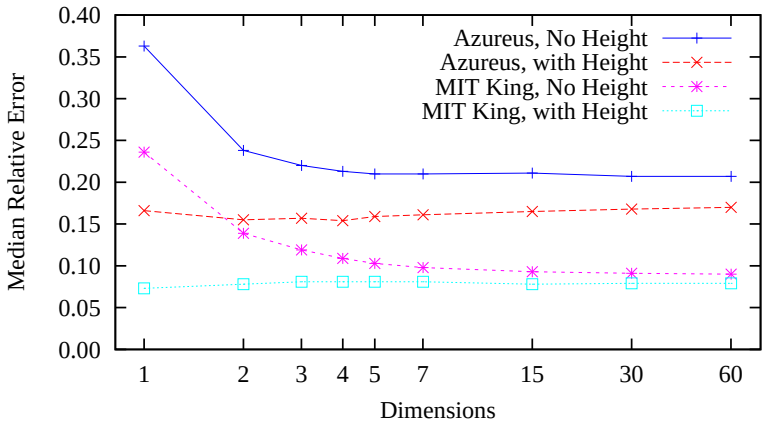

Figure 4.5: **Height.** Because *height* had a major, positive impact on Azureus in simulation, I returned it to the *4d+h* version.

---

lab, I compared the statistics from live Azureus nodes to those from simulated embeddings of the Azureus latency matrix. In Figure 4.7, I compare live and simulated relative error. The data show a significant gap between live and simulated performance. (Prior work using the same simulator found simulations of PlanetLab mirrored live results [46].) The medians of the relative error distributions are 26% and 14% for live and simulated coordinates, respectively, a difference of 45%.

The data suggest that network coordinates have been partially tamed, but can be made substantially more accurate, and, therefore, more useful for distributed applications that would like to make cheap, quick decisions between providers of the same service. I discuss the remaining major barriers to accuracy in the next section.

4.6 Barriers to Accuracy

In this section, I examine five primary causes of the remaining difference between the current live accuracy and what appears to be achievable based on simulation results. The five barriers are: churn, drift, intrinsic error, corruption, and latency variance. I present techniques that address the first three barriers and non-malicious corruption. However, malicious corruption and latency variance remain unsolved; indeed, the latter requires a fundamentally new approach to latency prediction. Based on my simulation and PlanetLab results and on monitoring Azureus over time, I have added the techniques that address churn, drift, and non-malicious corruption to the Azureus code. While preliminary experiments suggest they function as expected, I have not yet fully quantified their effects and do not include results for them here.

4.6.1 Churn

Distributed network coordinate algorithms traditionally consider churn as part of their network model. Researchers ask the question: given an existing, stable system, how quickly can a new node find a stable, accurate coordinate? Unfortunately, implicit in this question is the assumption that the existing system has converged, and this assumption breaks down in many large-scale distributed systems, including

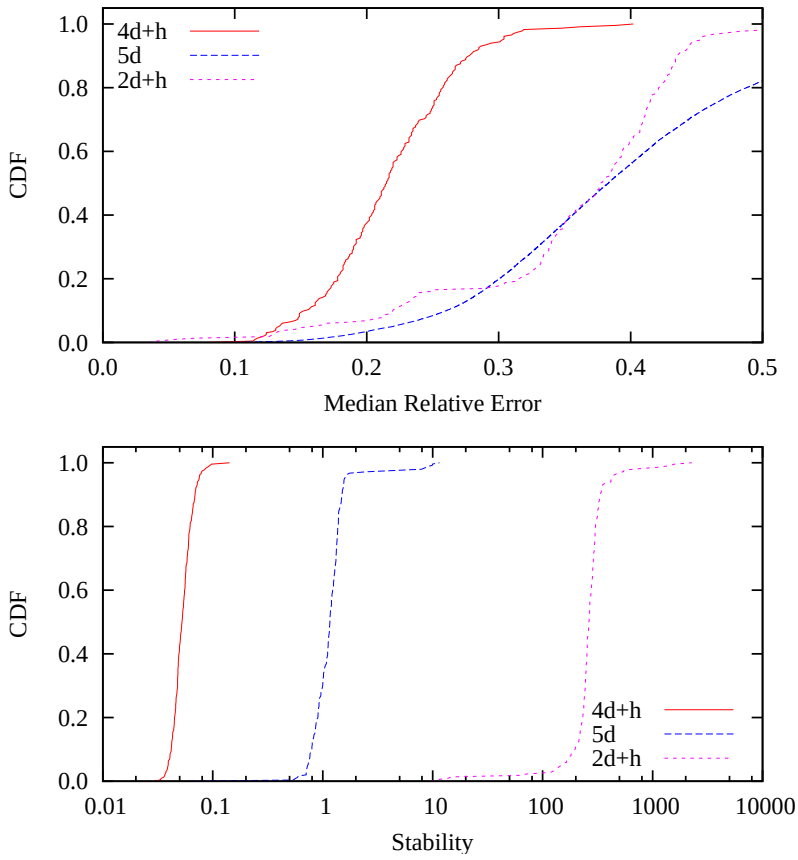

Figure 4.6: **Clients on PlanetLab.** The combination of filtering, neighbor decay, and height lead to substantially more accurate coordinates on PlanetLab nodes participating in the Azureus network coordinate system. Comparing 2D+H to 4D+H, the data show a 43% improvement in relative error and a four orders-of-magnitude improvement in stability.

Azureus. As Figure 4.8, illustrates, Azureus follows the long-tailed lifetime distribution typical of peer-to-peer systems [11]. (Azureus clients track uptime using an internal, query-able statistic.)

Because coordinate updates were on the order of tens of seconds or sometimes minutes apart, nodes often did not have much time to settle into a stable position before they exited the system. Using the data from the crawl of the live network, I separated nodes into ones that had been in the system for an hour or more and those that had not. I plot the relative error experienced by these two groups in Figure 4.9. The data confirm that these short-lived nodes, which make up the majority of the system, are substantially less accurate than long-lived ones.

Data on the mobility of Azureus's constituents is not available. However, the Azureus developers have anecdotal evidence that the trend follows general societal patterns of increasing mobility — that is, from server- and desktop-class machines to laptop- and cell-phone-class ones. As this trend continues, nodes in Azureus and similar systems will become increasingly transient.

I considered three potential solutions to the problem of sustaining a coordinate system under high

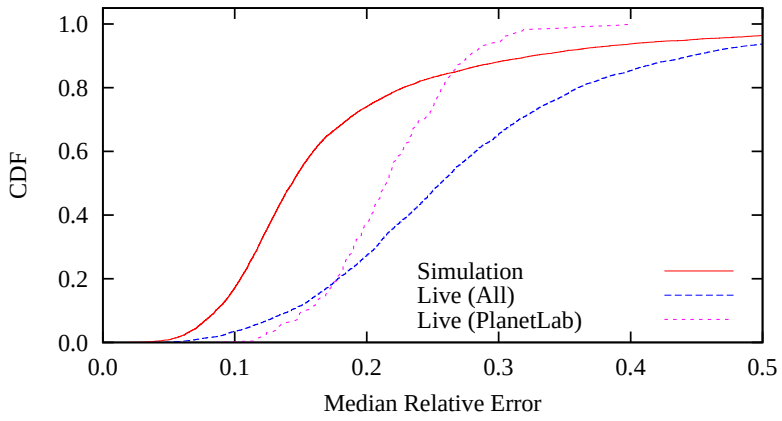

Figure 4.7: **Non-PlanetLab Clients.** Reality does not live up to expectations: a comparison of probed statistics from live Azureus nodes to those from simulation suggests that accuracy could be improved by as much as 45%. Section 4.6 explores the major remaining impediments.

---

churn rates. First, nodes could perform a rapid initial triangulation process before shifting to a lower update rate. However, adjusting the gossip rate over time has two problems: (a) "passive" (*i.e.,* maintenance-free) coordinate systems have no control over gossip and (b) in an "active" system, it would be a new, complex knob. Second, I considered "greedy optimization," where instead of just stepping once through the update process, nodes would repeat until a (local) minimum had been reached with respect to the currently known neighbors. Unfortunately, I found that this form of optimization does not work well until many neighbors are known, which is not the case early in a node's lifetime. Finally, I found a solution that is both extremely simple and had positive results in simulation: instead of starting from scratch when restarting a client, have it begin where it left off. I performed an experiment where I varied the amount of churn in simulation and toggled whether or not nodes "remembered" their coordinate on re-entry. In Figure 4.10, I show the results of this experiment. I found that when nodes started at the origin on re-entry, they had a deleterious effect not only on themselves, but on overall system convergence. In contrast, with this simple technique, accuracy remained about the same as when there was no churn. While this technique assumes limited drift (see next section), it appears to be a promising start to resolving the noxious effect of churn on live coordinate systems.

4.6.2 Drift

Monitoring my PlanetLab-based coordinate service over several months revealed that coordinates migrated in a fairly constant direction: the centroid of the coordinates did not move in a "random walk," but instead drifted constantly and repeatedly in a vector away from the origin. This was surprising because my previous study, based on a shorter, three-day trace, had not exhibited this pattern [46].

While coordinates are meant to provide *relative* distance information, *absolute* coordinates matter too. One problem with drift is that applications that use them often need to make assumptions on maximum distances away from the "true" origin. For example, one could use Hilbert functions to map coordinates into a single dimension [12]. This requires an *a priori* estimate of the maximum volume the coordinates may fill up. Mapping functions like Hilbert require that the current centroid not drift from the origin without bound.

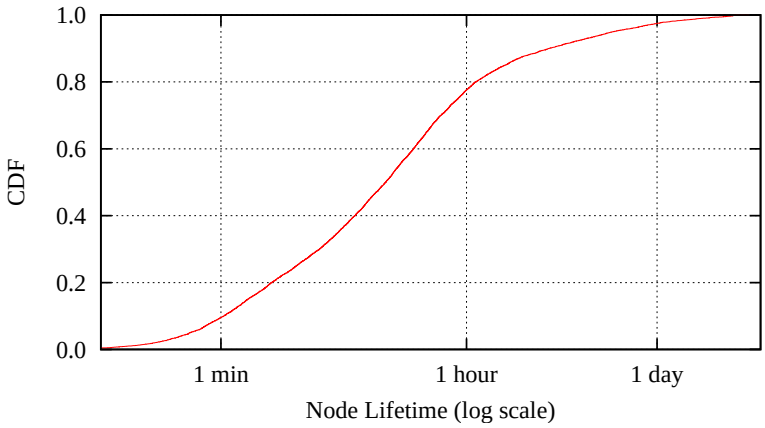

Figure 4.8: **Azureus client lifetimes.** Azureus nodes follow a typical peer-to-peer lifetime distribution curve. With 78% of its nodes in the system for less than one hour, it is difficult to incorporate the steady stream of newcomers with coordinates starting at the origin.

Gravity's ρ	Migration	Error
2^6	$8ms$	25%
2^8	$17ms$	10%
2^{10}	$74ms$	10%
2^{12}	$163ms$	10%
None	$179ms$	10%

Table 4.2: Small amounts of *gravity* limit drift without preventing coordinates from migrating to low-error positions.

Drift also limits the amount of time that cached coordinates remain useful [28].

A "strawman" solution to drift would be to continuously redefine the origin as the centroid of the systems coordinate. Unfortunately, this would require accurate statistical sampling of the coordinate distribution and a reliable mechanism to advertise the current centroid. My solution to drift is to apply a polynomially-increasing *gravity* to coordinates as they become farther away from the true origin. Gravity $\vec{G}$ is a force vector applied to the node's coordinate $\vec{x_i}$ after each update:

$$\vec{G} = \left(\frac{\|\vec{x_i}\|}{\rho}\right)^2 \times u(\vec{x_i})$$

where ρ tunes $\vec{G}$ so that its pull is a small fraction of the expected diameter of the network. Hyperbolic coordinates could use a similar equation to compute gravity.

Drift does not occur in simulation if one is using a latency matrix and updating nodes randomly, because this form of simulation does not capture time-dependent RTT variability. Instead, I used a 24-hour trace of my PlanetLab service to simulate the effect of gravity; we show the effect of different strengths of gravity in Table 4.2. The data show that this simple technique does keep the coordinate centroid highly stationary without affecting accuracy.

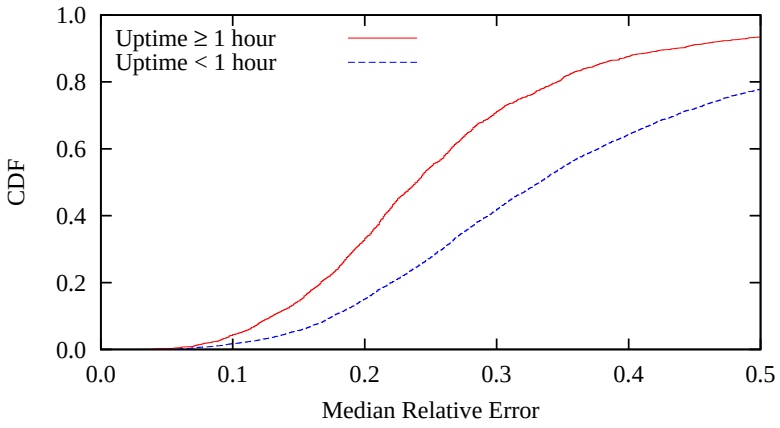

Figure 4.9: **Effect of lifetime on accuracy.** Azureus nodes that have been in the system for longer periods have more accurate coordinates. This suggests that churn may hurt convergence of Internet-scale coordinate systems.

To confirm the effect of gravity on a live system, we added it to my on-going PlanetLab service, which had ≈ 300 participants. In Figure 4.11, we compare drift before and after adding gravity over two 18 day periods. The data show that gravity effectively eliminates drift. In addition, it did not reduce accuracy, which, in both cases, had a median of about 10%. While gravity does not actively limit rotation, we did not observe a rate greater than one full rotation per three days. Determining the cause of drift is beyond the scope of this work.

4.6.3 Intrinsic Error

Violations of the triangle inequality occur more frequently and to a greater extent on Azureus than either on PlanetLab or for sets of DNS servers (see Section 4.3.3). I found, perhaps surprisingly, that removing a small number of the worst violators causes a large improvement in global accuracy. Not only do the violations these nodes take part in damage their own coordinates, but the damage they cause continues to reverberate throughout the system.

I performed an experiment where I removed a small percentage of the nodes with the largest triangle violations from the Azureus latency matrix and compared this to removing a random subset of nodes of the same size. I then computed a system of coordinates and found the relative error of each link. As Figure 4.12 illustrates, removing only the worst 0.5% of nodes leads to a 20% improvement in global accuracy. This data parallels results from theoretical work that showed how to decrease embedding distortion by sacrificing a small fraction of distances to be arbitrarily distorted [7]. These results show that *if* a mechanism could prevent these nodes from affecting the rest of the system, it would improve overall accuracy. Two example mechanisms for node self-detection and removal from the coordinate system are: (a) directly evolving an estimate of the extent of their violations by asking neighbors for latencies to other neighbors, and (b) determining if they are subject to traffic shaping (based on the modality of their latency distribution), and therefore a major cause of triangle violations. Preliminary experiments with self-exclusion based on bimodality tests show an improvement in accuracy of 8% at the 95^{th} percentile.

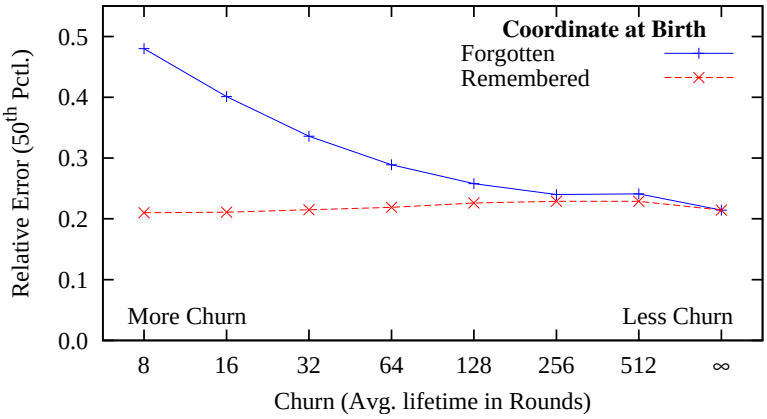

Figure 4.10: **Effect of coordinate recall.** Coordinate systems that experience high churn rates and do not allow nodes to "remember" their previous coordinates have trouble converging.

4.6.4 Corruption and Versioning

An insipid fact of running a large system where users can choose when to upgrade is that not everyone is running the same version. One of the problems I found with my original deployments was that about 13% of the remote coordinates received during gossip were at the origin; that is, $[0]^d$. After much discussion (Is that incredible churn rate possible? Do nodes behind firewalls never update their coordinates?), my collaborators and I realized that this problem was due to a portion of the network running an old code version. In fact, during one crawl of the Azureus network, I found only about 44% of the ≈ 9000 clients crawled were using the current version. While not very exciting, realizing this fact allowed us to compensate for it both in the coordinate update process and in active statistics collection through the explicit handling of different versions within the code.

Kaafar *et al.* have begun investigating the more interesting side of the problem of coordinate corruption: malicious behavior [38]. They model four distinct types of attacks and suggest *why* participants might want to behave badly. The attacks are disorder, isolation, free-riding, and system control. *Disorder* denies the service of accurate latency prediction by preventing coordinate system convergence. This can disrupt any service that relies on prediction and network positioning. In the case of Azureus, companies or groups that believe the network supports copyright infringement might wish to disrupt the network in this way. *Isolation* tricks a victim node into believing that an accomplice is the best node the victim should connect to for a particular service (*e.g.,* by appearing to be nearest). This allows the accomplice to monitor or alter the victim's overlay traffic. *Free-riding* reduces the attractiveness of the perpetrator, convincing victims it is distant and should not be used to host a service. *System control* is a generalization of *isolation*. Its goal is to trick all (or many) nodes about their network positions, again to monitor and alter victims' traffic.

In subsequent work, Kaafar *et al.* address *disorder* with a set of trusted landmark nodes that build their coordinates in isolation [37]. Nodes then compare their embedding to the nearest trusted landmark. If a node's embedding is found to be corrupted (*e.g.,* via a malicious neighbor), it forms a new set of neighbors and tries again. While I did not see any evidence of intentionally corrupt messages, it would be trivial to install a client, or a set of clients, that responded with random values, for example (just as the MPAA runs

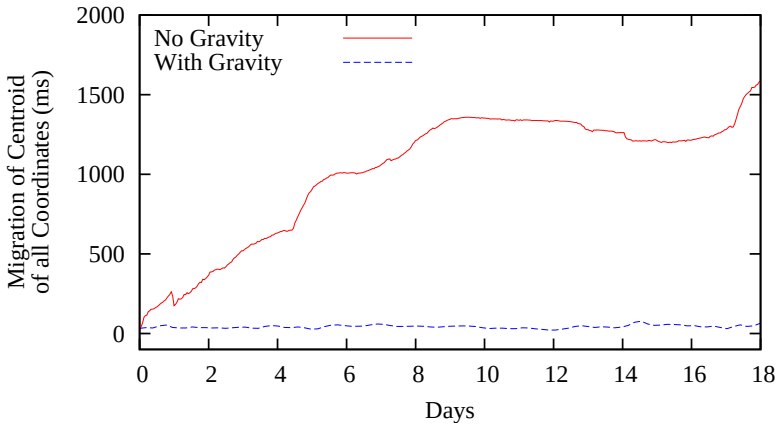

Figure 4.11: **Effect of Gravity.** With *gravity*, coordinates did not drift away from their original origin as they had done before.

---

clients with spurious content advertisements to squelch piracy). As Internet-scale coordinate systems come into wider use, they will need to grapple with both oblivious and malicious corruption.

4.6.5 Latency Variance

The prior "barriers to accuracy" paint a rosy picture; most problems have a fairly simple solution that practitioners can use to build more accurate, live coordinate systems. The existence of wide variation in latency measurements between the same pair of nodes over a short period of time is a harder problem with broad ramifications. If variances are very large what does it actually mean to "predict" the latency from one node to another? Using the data from the longest snapshot (5D), I determined the standard deviation of latency between each pair of nodes. I found that round trip times varied by a *median* of $183ms$. I show this distribution in Figure 4.13. This spread affects other latency prediction systems as well. A reactive measurement service, such as Meridian, will be more error-prone or have higher overhead if small numbers of pings do not sufficiently measure the latency to a high variance target. In fact, coordinate systems may be in a better position to address this problem because they can retain histories of inter-node behavior.

I applied *latency filters* to Azureus's coordinate implementation. They act as a low-pass filter: anomalies are ignored while a baseline signal passes through. Additionally, they adapt to shifts in the baseline that BGP route changes cause, for example. These filters assign a link a *single* value that conveys the expected latency of the link. While I found these simple filters worked well on PlanetLab, describing a link with a single value is not appropriate with the enormous variance I observe on some of Azureus' links.

I ran an experiment where I compared ICMP, filtered, and raw latency measurements that were taken at the same time. To determine which destination nodes to use, I started Azureus on three PlanetLab nodes and chose five ping-able neighbors after a twenty-minute start-up period. I then let Azureus continue to run normally for six hours while simultaneously measuring the latency to these nodes with *ping*. I plot the data in Figure 4.14. Figure 4.14 (a) illustrates a pair similar to the PlanetLab observations: there was raw application-level and ICMP variance, but a consistent baseline that could be described with a single value. In contrast, Figure 4.14 (b) portrays a high variance pair: while the filter does approximate the median round trip time, it is difficult to say, at any point in time, what the latency is between this pair.

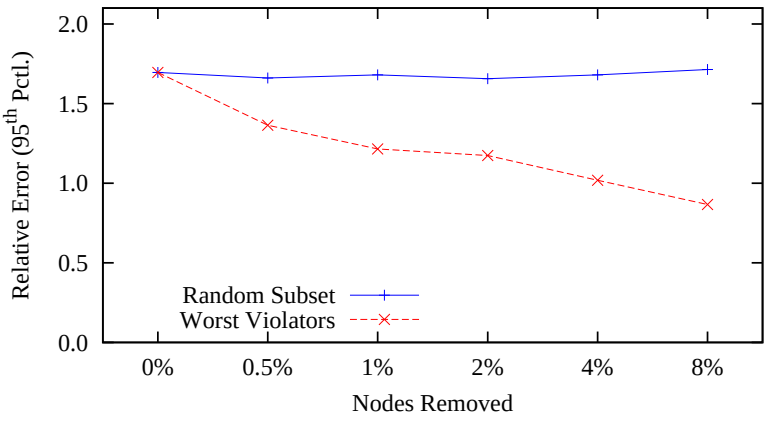

Figure 4.12: **Effect of Triangle Inequality Violations on Global Accuracy.** Removing only a small percentage of nodes with the worst triangle inequality violations has a large effect on global accuracy.

The impact of the dual problems of high latency variance and modifying algorithms to deal with high latency variance is not limited to network coordinate systems. Latency and anycast services deployed "in the wild" need to address this problem. While there may exist methods to incorporate this variance into coordinate systems — either through "uncertainty" in the latency filters or in the coordinates themselves — resolving this problem is beyond the scope of this thesis.

4.7 Summary

I have demonstrated that network coordinates in the wild do behave somewhat differently than do tame coordinates on PlanetLab or in simulation. Fortunately, even these wild coordinates can be tamed. While the initial network coordinate implementation illustrated some of the problems that critics often cite, I found that simple, but effective techniques overcame nearly all these issues. In Azureus, network coordinates provide a simple and efficient mechanism for anycast, as part of DHT lookups, and may soon be used to optimize streaming media. In addition to providing a wealth of data and analysis from a live, large-scale deployment, I have deployed and evaluated six techniques that improve the accuracy and/or stability of network coordinate systems: latency filters, update filters, neighbor decay, coordinate memory, gravity, and violator exclusion. Together, these yield efficient, accurate, and stable network coordinates in the million-node Azureus network.

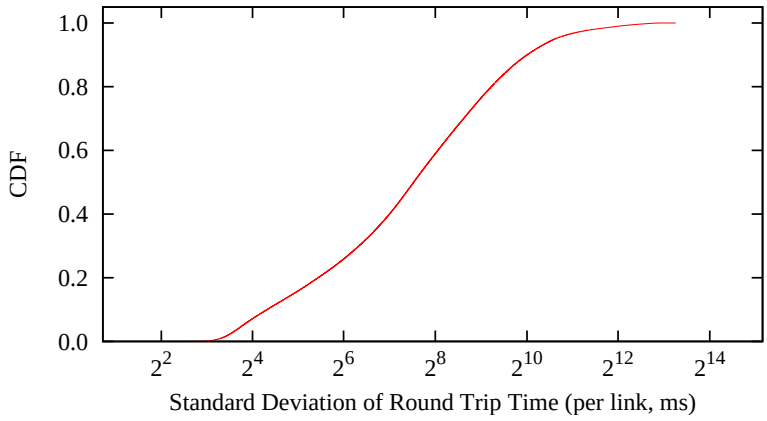

Figure 4.13: **Round Trip Time Variance.** When round trip times vary by a *median* of $183ms$, what does it mean to summarize a latency prediction with a single value?

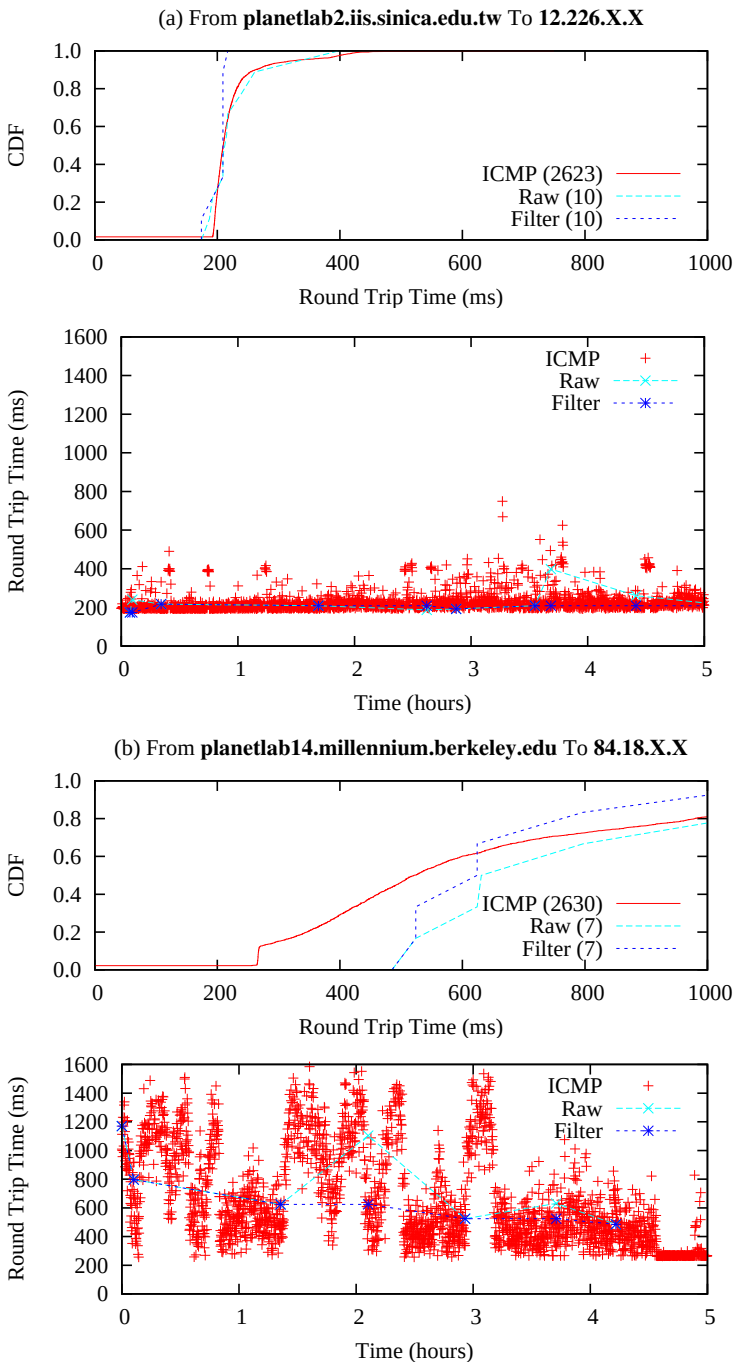

Figure 4.14: **Round Trip Times for Two Pairs of Nodes.** A comparison of round trip times between two sets of node pairs using ICMP, raw application-level measurements, and filtered measurements. Pair (a) exhibits some variance, but shows a consistent baseline. With pair (b), the variance is so large that assigning this node a coordinate — or putting it into a consistent Meridian ring — is bound to be an error-prone process. The number in parentheses in the legend is the number of round trip time measurements in the cumulative distribution function.

Chapter 5

Wired Geometric Routing

Routing substrates are an important building block for large distributed applications. Existing overlay routing substrates, such as distributed hash tables, ignore node locality because they are based on a random identifier space. This can lead to lower performance for locality-sensitive applications, such as web caching, distributed gaming, and resource discovery.

This chapter examines the challenge of building a locality-aware routing substrate on top of a locality-based coordinate system, where the distance between coordinates approximates network latencies. As a starting point, I take an existing algorithm designed for efficient routing in a two-dimensional Euclidean space. I address the practical problems of forming routing tables with imperfect node knowledge and churn and examine query performance on non-Euclidean data sets — those seen in real-world latency distributions.

Since this chapter was first published, one of students of a co-author (Pietzuch) has begun creating an implementation based on the simulator and techniques we developed here. This implementation will become part of the Pyxida open source network coordinate library.

5.1 Introduction

Routing is a basic, necessary primitive for any distributed system. Over the previous decade, overlay networks based on the random key paradigm (*e.g.,* [74]) have become ubiquitous; they have moved from the research playground to being the core of several systems with millions of users (*e.g.,* [6, 24]). Routing with random keys is simple: destinations are defined in the same key namespace as nodes; each node stores routing entries that refer to other nodes across the key space; and nodes forward messages to the neighbor whose key is closest to the target key.

However, the performance of applications, such as distributed web caching, scalable anycast, and resource discovery, often depends on the latency and reliability of the paths taken by messages. For example, in a distributed multi-player game, it is important to host the game server at a node close to the centroid of all client locations to ensure fairness in access latency. Similarly, a web caching infrastructure will exhibit low latency when client requests are routed to the closest existing cache.

While the random key paradigm has many intrinsic benefits, such as load-balancing and resilience to churn, its starting point is to intensionally ignore physical geography and node location. It then boosts performance with proximity-aware techniques, such as preferring nearby nodes to distant ones when they can perform the same role in message forwarding [70]. These proximity-aware overlays do not rely on locality for correctness but use it only to improve network efficiency.

In contrast to this dominant body of work on random key routing, a series of theoretical papers examine efficient routing schemes for graphs where the vertices are points in the Euclidean plane [2, 31, 43, 51, 78]. These locality-aware designs depart from a logical identifier space, creating overlay topologies that are based purely on physical node distance. The most recent of these, Abraham and Malkhi's work [2], makes explicit the connection between routing on these graphs and network coordinates: the vertices are the embedded positions of nodes in the network. This previous work examined efficient algorithms for routing on top of a coordinate substrate within a theoretical framework. This chapter begins where this more abstract research left off.

In this chapter, I take a first look at both the practical problems associated with coordinate-based routing and the performance one could expect from deployment on a distributed system. In particular, I examine the challenges of (a) forming and maintaining locality-aware routing tables, (b) routing to the node *nearest* a coordinate (the analog of finding the root for a distributed hash table query), and (c) the primary parameter tradeoffs in coordinate-based routing.

This chapter is organized as follows:

Section 5.2 provides additional background on the problem of overlay routing in wired overlay networks.

Section 5.3 describes $\hat{\theta}$-routing, the algorithm for Euclidean plane routing upon which I build.

Section 5.4 addresses the central challenges for this work to be applicable in a real network setting, for example, by proposing a construction for routing tables through a join protocol and background gossip.

Section 5.5 evaluates the performance of routing in a d-dimensional coordinate space using simulation; in particular, I analyze the state-accuracy trade-off and accuracy under churn.

Section 5.6 summarizes the results of this chapter.

5.2 Background

Routing techniques that take advantage of network geometry first arose in the context of *wireless networks* [9, 26, 39, 48]. In a wireless network, deployed nodes are capable of communicating directly via radio communication only to nodes in their vicinity. Taking advantage of a node's local knowledge of its neighbors and a "sense of direction" for routing decisions results in efficient routing algorithms. Fundamentally, such wireless geometric routing is locality-preserving because the network does not permit long-distance hops due to the limited range of radio links.

To apply similar ideas to *wired overlay networks*, nodes must obtain knowledge of their location within the network. In the routing algorithm in this chapter, nodes will equate their position in the network with their network coordinate. This position is based on communication latencies, as opposed to true geographic position. This method for assigning position is useful because it encapsulates latency estimates to neighbors, reflecting the delays caused by overlay message routing.

Wired overlay networks support long-distance hops for direct communication between nodes outside of each others' vicinities. Both the lack of knowledge of one's immediate neighbors and the potential for intelligently placed long-distance edges make routing in the wired and wireless domains significantly different.

I consider the problem of wired geometric routing as follows. Given (1) that nodes have coordinates that represent their location and (2) each node has a *routing table* containing a small set of other nodes, I want an efficient method to (a) construct routing tables and (b) route messages to a target location

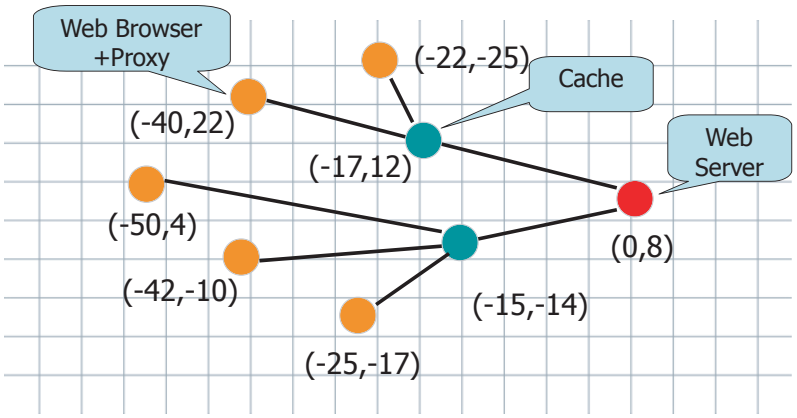

Figure 5.1: **Locality-aware Web Cache.** If all members of a system are assigned network coordinates that predict latency, clients can easily find nearby caches by routing *toward* the web server's location. Clients, that could host proxy software to direct traffic and a standard web browser, would route requests to the desired web server. Requests would hit the nearest overlay cache. Clients in the same neighborhood would tend to hit the same cache because their routes would converge.

---

via multiple hops. A target location may be either the exact coordinate a node or a coordinate for which the (approximate) closest node should be found. This *approximate nearest neighbor routing* facilitates the implementation of many locality-dependent applications, such as a distributed web cache as illustrated in Figure 5.1.

The recent theoretical work on Euclidean routing algorithms described in the next section provides the foundation for coordinate-based routing. However, a number of interesting questions remain to be solved: (1) how can routing tables be constructed in a decentralized manner? (2) how can they be maintained under churn? (3) how do the existing methods need to change to support nearest neighbor queries? (4) how does the error inherent in network coordinates affect the utility of routing results? I build on this prior work to answer these questions, highlighting the practical issues with wired geometric routing, and I examine what can be expected of its performance in realistic settings.

5.3 Scaled θ-routing

Several proposals exist for efficient geometric routing in a two-dimensional Euclidean plane [2, 31, 43]. I focus on a method called scaled θ-routing because its assumptions make it directly applicable to wired overlay networks with network coordinates. In particular, it assumes that (1) each node has a coordinate in a Euclidean space and (2) nodes do not have a maximum communication range: any node may connect with any other.

Hassin and Peleg Scaled θ-routing [31] is based on Keil and Gutwin's θ-routing [41]. θ-routing takes advantage of the "sense of direction" in a Euclidean plane by routing "towards" a given target, and is based on the construction of a θ-graph spanner. Such a graph has the property that every pair of nodes is connected by a "short" path. More formally, for any two nodes n_1 and n_2, the length of the routing path (n_1, n_2) is no more than a constant factor times their Euclidean distance $|n_1 n_2|$.

The construction of a two-dimensional θ-graph spanner is simple. Each node subdivides the area

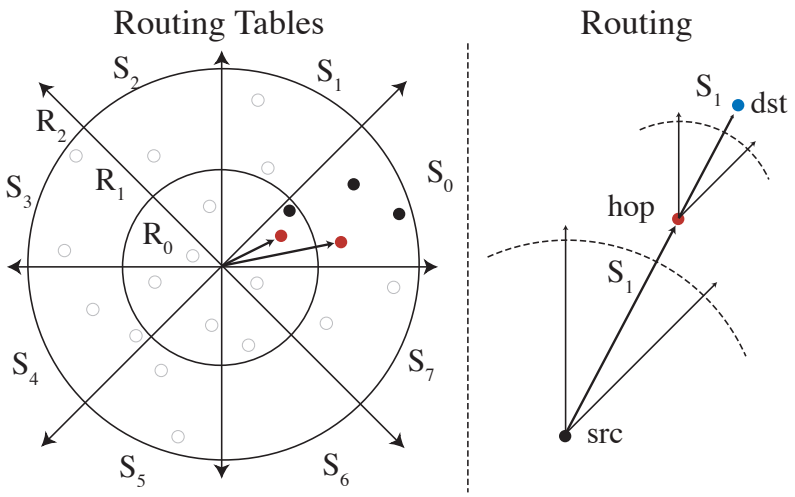

Figure 5.2: **$\hat{\theta}$-routing Sectors and Rings.** $\hat{\theta}$-routing subdivides the coordinate space into $\frac{2\pi}{\theta}$ sectors and r rings. The left figure illustrates which nodes are selected for routing table entry in sector S_0 for rings R_0 and R_1 in a network where $\theta = \frac{\pi}{4}$ and $r = 3$. The right figure portrays routing from a source *src* to a destination *dst* via one hop. At each step, the zone of the destination is calculated and the message is greedily forwarded to the furthest hop that does not exceed the target's zone.

---

around it into $\frac{2\pi}{\theta}$ *sectors* of angle θ. For each sector, it inserts the nearest neighbor into its routing table. Messages are forwarded by sending them to the neighbor in the same sector as the target location. The advantages of θ-routing are that the routing table size is small and routing paths have low delay stretch, which is the relative penalty of the route taken compared to direct IP routing (because they hold closely to the line between the source and the target). However, it suffers from a linear hop count that is proportional to the diameter of the network.

Hassin and Peleg [31] improve on the linear hop count of θ-routing by adding long-distance links to traverse large distances in a single hop. In *scaled θ-routing*, abbreviated as $\hat{\theta}$-routing, the θ-graph spanner is augmented with exponentially expanding rings. To create its routing tables, each node partitions the space around it into $\frac{2\pi}{\theta}$ sectors and r rings. The areas obtained through the intersection of sectors and rings are called *zones*. Nodes keep as neighbors the nearest node that they know in each zone. I illustrate $\hat{\theta}$-routing in Figure 5.2. Compared to *θ-routing*, *$\hat{\theta}$-routing* reduces the hop count to logarithmic complexity, but increases the amount of routing state each node maintains.

5.4 Practical Wired Geometric Routing

Given Hassin and Peleg's algorithm and given the ability to construct stable and accurate live network coordinate systems, I needed to address numerous practical barriers that apply to $\hat{\theta}$-routing and other geometric routing proposals. In this section, I highlight three main challenges to coordinate-based routing and my proposals for how to address them. The challenges are: (a) mapping nodes to zones when $d > 2$, (b) efficiently learning neighbors for routing table construction, and (c) performing nearest neighbor queries instead of ones with a known target coordinate.

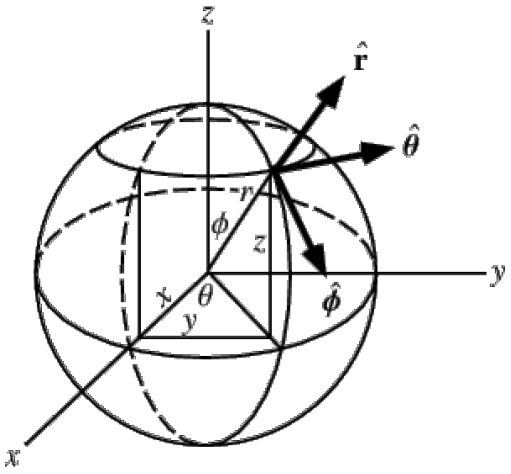

Figure 5.3: **Zone Assignment in d dimensions.** Because network coordinates have lower prediction error with $d > 2$ dimensions, it was important to generalize $\hat{\theta}$-routing with hyperspherical coordinates. With hyperspherical coordinates, each dimension "slices" the sectors of prior dimensions; thus, the total number of zones grows exponentially with the number of dimensions.

5.4.1 Zone Assignment

Figure 5.2 illustrates the intuitive process of assigning neighbors to sectors in a system with two dimensions: given the angle θ_i of the neighbor i and a set of rays each θ degrees apart, one simply finds which rays θ_i lies between. However, this process proved surprisingly challenging in three or more dimensions.

A straightforward example in more than two dimensions is the three dimensional case when $\theta = \frac{\pi}{2}$. Here, one can readily visualize that there are 8 sectors with the axes acting as the rays that divide each sector. In addition, the assignment of neighbors to zones is fairly intuitive: one can find the spherical coordinates θ and ϕ and perform essentially the same method of assignment as done in two dimensions. Figure 5.3 shows the relationship between the Cartesian vector (x, y, z) and the angles θ and ϕ. Note that one of the two angles will range from $(-\pi \ldots \pi)$ and the other from $(0 \ldots \pi)$; one completes the full circle but the other needs to traverse only half that to cover all possible directions. Each angle can then be dealt with separately, resulting in four possible sectors from the angle with the larger domain and two from the one with the smaller.

While harder to visualize, this method generalizes to an arbitrary number of dimensions. It is simpler to operate on Cartesian coordinates for network coordinate refinement processes such as Vivaldi [19]. For zone assignment, however, hyperspherical coordinates $\phi_0, \ldots, \phi_{d-1}$ are more intuitive. For the assignment, the d-dimensional vectors from each node to its neighbors is converted into its hyperspherical equivalent. Because each additional dimension effectively "slices" the sectors of prior dimensions, the total number of zones increases exponentially with the number of dimensions to $2 \times s \times s^{d-2}$, where s is the number of sectors per standard dimension.

5.4.2 Maintenance Protocol

The routing algorithms described in Section 5.3 make greedy routing decisions — that is, without global knowledge or knowledge of the prior path of the query. However, they rely on omniscient assignment of neighbors to routing tables. This assumption of global knowledge needs to be removed for use in a distributed setting.

In addition to constructing routing tables, an important question is how does geometric routing perform in the absence of perfect information: does it fail gracefully or catastrophically? For $\hat{\theta}$-routing, there are three possible error conditions for a given zone: (1) a neighbor is not the truly nearest node in that zone, (2) a neighbor has failed, or (3) there is not an entry for a neighbor when one exists. Intuitively, the most important entries in the routing table are the nearest nodes in each sector. With only these edges, routing proceeds correctly, just with a higher hop count. This has an analog in one-dimensional DHT routing: routing succeeds as long as every node knows its direct successor in the namespace. Thus, acquiring and maintaining this local knowledge first and then optimizing long-range links second seems a sensible approach to constructing neighbor tables for $\hat{\theta}$-routing.

My maintenance protocol makes several assumptions: (a) a simple periodic gossip mechanism exists that allows nodes to exchange routing tables; performed at random using an existing neighbor from the table, this strategy eventually leads to knowledge of one's local neighbors given enough time and limited churn; (b) each node bootstraps into the system with a single randomly chosen neighbor; (c) dead nodes are discovered through failed routes or failed gossip.

Given the benefit of focusing on local neighbors primarily, I borrow a simple technique from Pastry [70]: instead of exchanging routing tables with a bootstrap neighbor, a node performs a query for its own coordinate which begins at this neighbor. This special query is flagged so that each node on the path adds its routing table to the message. When the message is eventually delivered to the node, it is likely to contain at least one nearby neighbor, whose routing table will be similar to what the new routing table should be.

I define the performance metric *local knowledge* to be the fraction of sectors for which each node has correctly identified its nearest nodes, averaged over all nodes. I have found experimentally that this metric is strongly correlated with routing accuracy, that is, correctly reaching a target. One could imagine a more complex metric of *weighted local knowledge* for which a gossip strategy could optimize: instead of attempting to learn only of the nearest nodes in each sector, the strategy would aim to learn of the larger vicinity, but give more weight to a node's immediate locality. Both metrics could be refined based on the number of sectors and the distance of each neighbor. I examine the effect of the maintenance protocol on local knowledge in Section 5.5.3.

5.4.3 Types of Queries

Previous work on coordinate-based routing algorithms make an assumption that is impossible to attain in a dynamic distributed system: that the source of each query knows precisely the target node's current coordinate. Because node locations change and coordinates are therefore constantly under adjustment, one cannot assume that a target node's coordinate remains unchanged even over short periods of time. Thus, without perfect knowledge, all queries are in fact nearest neighbor queries. In addition to queries for a target node's coordinate, applications will want to perform queries for a computed location, such as the centroid of a set of points. In this case, the goal of the query is to find the closest overlay node to the computed point. A final type of query is to broadcast a message to all nodes within an abstractly defined region (*e.g.,* a radius). Figure 5.4 illustrates these three qualitatively distinct forms of queries.

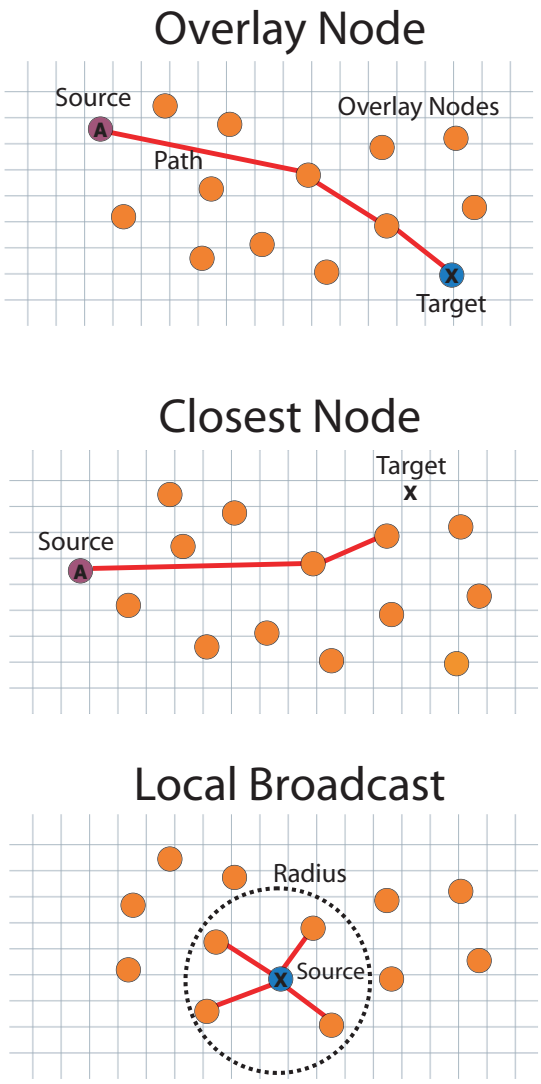

Figure 5.4: **Types of Geometric Routing.** There are three qualitatively different types of geometric routing queries: (a) Overlay Node Query: route message to overlay node at location X; this is analogous to *route(key,msg)* in DHTs, but the routing path between A and X has low latency and physical meaning; (b) Find the Closest Overlay Node to a Target: used when location is external to overlay network (*e.g.,* finding closest web crawler to a web server with location X) and when the location has been computed (*e.g.,* the centroid of overlay nodes' coordinates); (c) Local Broadcast: send a message to nodes in a neighborhood, which could be defined by a radius or other boundary (*e.g.,* gossip across local web caches).

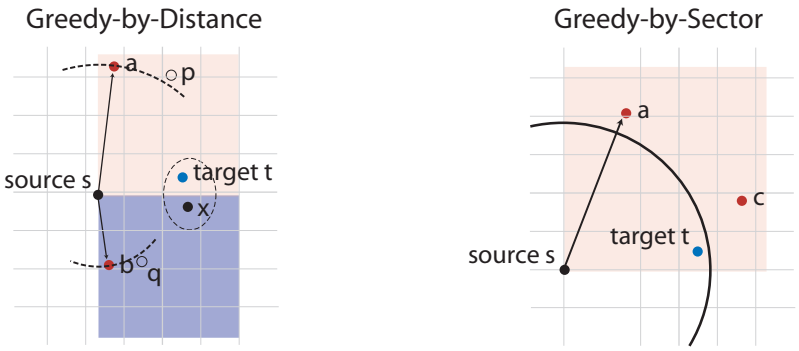

Figure 5.5: **Local minima in Closest Node Queries.** Minima can occur even with "perfect" routing tables, where each node links to the nearest node in each of its zones. Two situations where $\theta = \frac{\pi}{2}$ are shown. With Greedy-by-distance (left), with a source s and a target t, the path halts at s because it is nearer to the target than any of its neighbors. Even if s stored its neighbors' routing tables (as in [55]), it would remain stuck assuming $\|st\| \leq \{\|at\|, \|bt\|, \|pt\|, \|qt\|\}$. Greedy-by-sector routing would continue on to the correct nearest node x. However, Greedy-by-sector (right) routing can halt one hop from the true nearest node, where $\|ct\| < \|st\| < \|at\|$ and $\|sa\| < \|sc\|$. Because node c is not in the routing table of node s, node s is unaware that node c, the closest node to t, exists.

Broadening the types of queries from those where the target *is* a node and therefore must exist in at least one node's routing table (assuming "perfect" tables) has some interesting ramifications. With a nearest neighbor query, it is difficult to know when the true nearest neighbor has been reached — when to terminate the query. In particular, I found that an obvious optimization to $\hat{\theta}$-routing frequently foundered on nearest neighbor queries. $\hat{\theta}$-routing is intended to operate greedily on sectors; that is, forward to the neighbor that covers the maximum distance to the target while remaining in that sector. There may exist a node, not in the same sector as the target, but much closer to it than the in-sector node. Exclusively following this greedy-by-distance approach, however, can lead to local minima several hops from the target and frequently did so in experiments. Greedy-by-sector — where the hop selected is always from the target's sector, if possible — can also lead to falling one hop short of the true nearest neighbor, but this occurred in 0.0001% of trials in our experiments using network coordinates derived from real latencies. Figure 5.5 illustrates these two cases. In practice, I only used greedy-by-distance routing if no neighbor existed in the target's sector. Guaranteeing that the true nearest neighbor is found is beyond the scope of this work.

5.5 Results

I investigated three main aspects of geometric routing through an implementation of $\hat{\theta}$-routing: (a) how the sector and ring parameter space particular to $\hat{\theta}$-routing affects routing performance in an abstract Euclidean graph; (b) how network coordinates derived from a real world data set affect finding the closest overlay node to a target; and (c) how the maintenance method we proposed in Section 5.4.2 performed under churn.

I implemented $\hat{\theta}$-routing in a simulator that allowed us to easily vary its central parameters, inter-node latencies, and sets of network coordinates. I assume there is no message loss and that inter-node

latencies are static during the experiments.

5.5.1 Sector and Ring Parameters

Hop count and delay stretch are important metrics for any overlay routing strategy. I set up an experiment that captured the theoretical model for $\hat{\theta}$-routing: all nodes had "perfect" routing tables and queries were for a participant's true coordinate (they were not closest node queries). In Figure 5.6, I show results from the two-dimensional case where 1024 coordinates were chosen from a unit square.

The sector and ring parameters have distinct, interesting effects. The data show that with long edges (large r) and few sectors (large θ, *e.g.,* $\frac{\pi}{2}$), routes zig-zag towards the target, resulting in a high delay stretch. As the number of sectors increases, each hop veers less from the direct line between the source and the target, diminishing stretch. But because increasing the number of sectors increases state exponentially, a decision to use small angles with $d > 2$ can cause high overhead. Because the dimensionality of network coordinates tends to range from three to about seven, very small angles may not be a practical option.

A base b determines the exponentially-increasing radius of each ring. With a unit square, a base of $\frac{1}{1000}$ expands to cover the network after twelve rings, but because the average distance between nodes is $\frac{1}{3}$, the first few rings are of little benefit because they seldom contain any nodes.

A further complication with choosing a good sector, angle, and base is the shape of the network. I have found that, in network coordinate systems composed of Internet nodes, one or two dimensions are dominant; they capture the physical distance across the Earth's surface while remaining dimensions serve as "wiggle room" that minimizes error (see Section 4.3.4). A good parameter choice in one dimension may not be appropriate across all; designing parameters that vary based on dimension may be appropriate in practice and is an issue for future work.

5.5.2 Closest Node Queries with Network Coordinates

While the performance of $\hat{\theta}$-routing routing on a hypercube confirms my intuition on the roles of different parameters, it does not illuminate what to expect from wired geometric routing on network coordinates derived from the Internet. In addition, the previous experiment does not suggest how closest node queries — not queries for active participants — perform.

To distinguish between the exigencies brought about by real network coordinates and those brought about by churn and other causes of partial network knowledge, in this experiment nodes are again assigned their "perfect" routing tables. I created a low-error (median relative error $\approx 6\%$), four-dimensions-plus-height embedding from the inter-DNS server trace from Dabek *et al.* [19] gathered with the King method [29]. In this trace the average latency between node pairs is $180ms$ and the network diameter is $800ms$. I performed closest node queries by designating 10% of the 1740 nodes as non-participant targets. I let the number of rings $r = 8$, the base ring radius $b = 4ms$, and the sector angle $\theta = \frac{\pi}{3}$.

The results of these experiments showed that, although all queries found the closest or second closest node, this node was often not the truly nearest node to the target in terms of latency. Because queries were finding the node with the correct coordinate, almost all of the latency penalty an application would experience using that node instead of the true nearest was due to the error inherent in the embedding process. Because queries took on average 4.18 hops, a simple, inexpensive optimization to this process was for each node on the routing path to measure its latency to the target and then to report the node with the lowest latency as the nearest node at the end of the query. I plot the results from $50k$ queries in Figure 5.7. The results show that this technique reduced the absolute latency penalty by 43% at the median (to $7ms$).

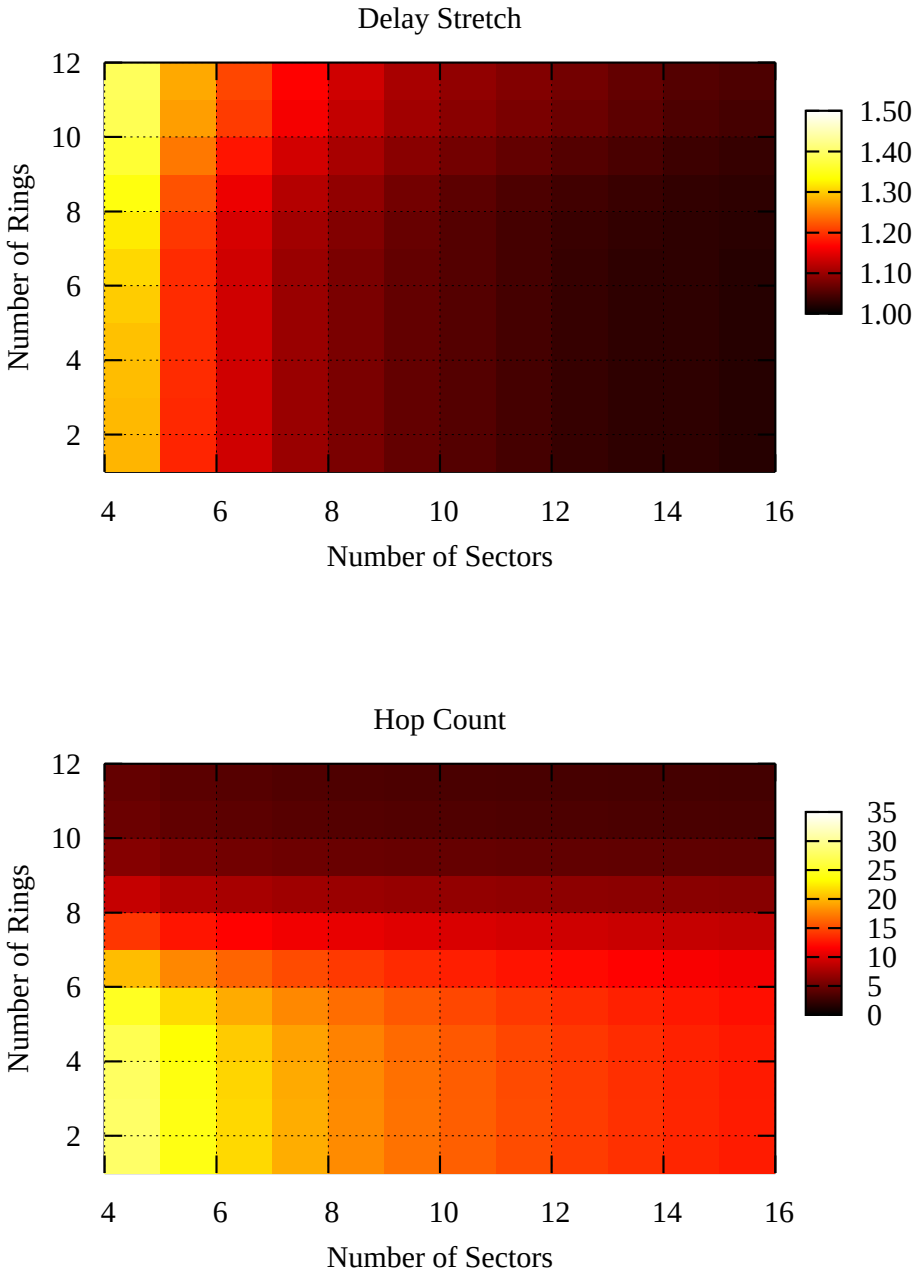

Figure 5.6: **"Perfect" Routing Tables.** For two-dimensional routing with "perfect" routing tables, far reaching rings and few angles result in queries that zig-zag across the coordinate space. Additionally, ring radii need to be tuned to the size and shape of the network to reduce hop count.

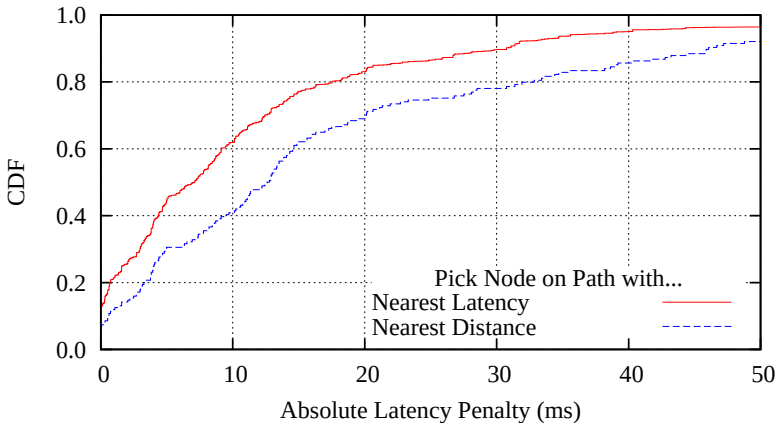

Figure 5.7: **Closest Node Queries on real-world network coordinates.** For both the standard ("nearest distance") and optimized ("nearest latency") strategies, the node returned by a typical query is close to the target in terms of latency. For a target node t, a query result node q, and the node p with the lowest latency to t, the absolute latency penalty is $l(q,t) - l(p,t)$.

Because the latency penalty is so dependent on the embedding error, we anticipate that any improvement in coordinate accuracy will further improve these results.

5.5.3 Finding Nearest Neighbors under Churn

My final question was whether the process of each node discovering its inner ring of nearby neighbors could be sustained under the more realistic setting of node arrival and departure. To simulate churn, I set nodes' lifetimes to a Poisson-distributed random variable and varied the mean. I assumed nodes gossipped with a neighbor from their routing table once every five minutes and let them either bootstrap using a random node or follow the Pastry-like join mechanism described in Section 5.4.2.

In Figure 5.8, I show the 80^{th} percentile absolute latency penalty (again compared to the node with the lowest latency to the target). The data exhibit two interesting characteristics. First, the join protocol reduces the latency penalty by 34% at the highest level of churn. Routing succeeds at this churn rate because nodes typically have knowledge of 54% of their immediate neighbors (their local knowledge). Second, as the churn rate falls, the join technique lessens in importance; again this is reflected in their local knowledge metric: it falls to 36% with average lifetimes of two hours. In fact, I found a very strong correlation between this metric and the latency penalty ($R^2 = 0.87$). Together these characteristics suggest that the routing table state may be significantly optimized with little or no loss in accuracy as long as a strong set of links to immediate neighbors is kept.

5.6 Summary

In this chapter, I presented a practical locality-aware routing substrate that takes advantages of recent advances in algorithms for routing in a Euclidean plane. I addressed important challenges, such as routing table construction under imperfect knowledge, maintenance protocols for new nodes, and the support for nearest neighbor queries.

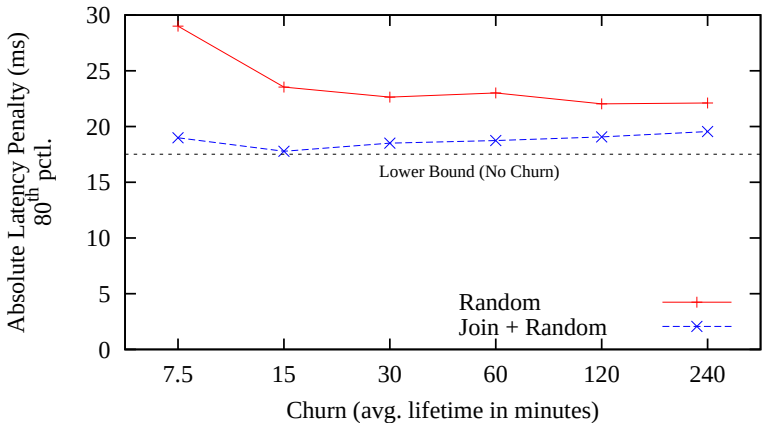

Figure 5.8: **Closest Node Queries under Churn.** Using the Pastry-like join protocol leads to more accurate routing tables than random periodic gossip, particularly in systems with high churn rates (As Figure 4.8 shows, in Azureus 78% of its nodes are in the system for under an hour.) The lower bound line is taken from the previous experiment where there was no churn.

In my simulation study of routing on network coordinates, I found that successfully discovering the true nearest neighbor to a node (in terms of latency) is almost entirely dependent on the accuracy of the underlying network coordinates; any improvement to these in turn will directly improve nearest neighbor accuracy. Thus, under relatively static cases, trade-offs in the state space of the routing algorithm can be made independently of the particular network. Under cases with churn, maintaining a valid connection to one's nearest neighbor in the immediately adjacent zones is important for routing correctness. There exists a strong correlation between knowing nodes in the local vicinity (the *local knowledge* metric) and finding the correct nearest neighbor under churn. My join technique works well under high churn (when nodes are using it frequently), but a method to maintain strong local knowledge is needed to keep local knowledge strong on an on-going basis.

This chapter provides the first example of efficient routing using a network coordinate substrate. This form of routing is qualitatively different from random key routing with proximity-awareness. More broadly, coordinate-based routing is an essential building block for higher level abstractions such as multicast and remote service discovery. For example, nodes that wish to discover a service in a given location would first route to the coordinate of that location, then use one of several techniques, such as an expanding ring search, to complete the search, finding a node running the service (or one capable of running the service) within the vicinity. With the implementation of this work as part of the Pyxida project, these higher level locality-aware services can now be built.

Chapter 6

Locality-Aware Anycast

Through latency prediction, network coordinates can add locality-awareness to applications. In turn, locality-aware applications make better use of the resources available to them, improving application performance. In this chapter, I show how two applications use network coordinates to select resources from among many choices. These resource selection decisions are forms of *anycast*, where many nearly-equivalent instances of a resource exist. The coordinates help efficiently select a "good" instance from among these instances, which may differ in quality of service in terms of latency, bandwidth, load, or availability. In both of this chapter's applications, network coordinates provide a rapid, low overhead method for performing the anycast. While I show that network coordinates do facilitate anycast decisions, they are not a complete resource discovery solution. The chapter concludes with a qualitative discussion of which types of applications make good targets for network coordinates and which do not.

6.1 Introduction

In previous chapters, I have shown that network coordinates provide a scalable mechanism for predicting latencies in wide-area networks. This ability to predict is more a means than an end. Questions remain on the end-to-end benefit to applications — a more useful goal — and on the circumstances where network coordinates are the right tool. This chapter answers these questions.

In this chapter, I measure the application-level benefits of network coordinates and discuss under what contexts they are the appropriate solution. I focus on two applications that use network coordinates to perform anycast decisions, decisions that quickly and cheaply make a locality-aware choice given many possibilities. In the first application, distributed hash table (DHT) traversal, nodes use network coordinates to quickly select the closest node to themselves from a set without performing any extra measurement. DHT traversal is at the core of many distributed mechanisms, including put/get (hash table functionality), key-based multicast (publish/subscribe functionality), and other higher-level services. In the second application, swarm localization, a centralized server in BitTorrent uses network coordinates to determine the relative proximity of large collections of nodes, creating a more efficient mesh than BitTorrent's current random graph. Making these meshes more efficient both improves user download times and reduces overall bandwidth consumption. The latter is particularly important because BitTorrent constitutes at least 18% of total Internet traffic [60]. Network coordinates, however, are not the right tool for all circumstances. Not only are many criticisms of them well-founded, but, in addition, they can only constitute a component of a resource discovery framework. I consider the positive and negative criticism network coordinates have received and draw conclusions on the circumstances where their use is appropriate.

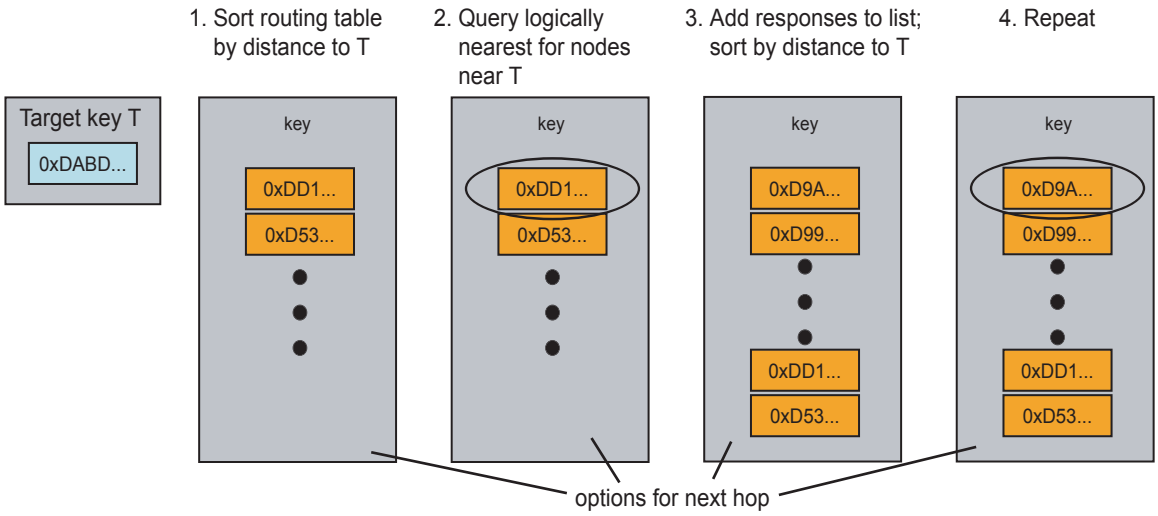

Figure 6.1: **Example of Kademlia Traversal.** The query source node discovers nodes progressively closer to a given target key until it finds a node storing the key or learns that the key/value pair does not exist. Step 1: the query source node sorts its own routing table by the logical distance to the target key T. Step 2: the source node queries the logically nearest node as its first hop, asking for nodes whose keys are closer to T than the source node. Step 3: the source takes the hop's response, adds it to its own list, and, Step 4, begins again. Because each hop makes logical progress toward the target key, the query is guaranteed to terminate.

6.2 Distributed Hash Table Traversal

In this first application, the central goal is to minimize delay when performing a lookup in a distributed hash table. Distributed applications use DHTs for storage [17] and publish/subscribe [13], which in turn serve as components of their own. Because routing is at the core of these higher-level operations, minimizing traversal delay improves user-perceived response time.

I examine DHT traversal within the Azureus application. Azureus uses a DHT for tracker announcements, sharing torrent ratings, and tunnelling through network address translation layers. Because Azureus's constituents are highly transient, it employs only limited caching; instead, frequent probes of the DHT ensure data is up-to-date. A typical user's session involves hundreds or thousands of traversals.

Azureus uses the Kademlia algorithm as the basis for its distributed hash table [52]. Kademlia overlay routing structure is similar to other DHTs, such as Chord [74] and Pastry [70]: each node has a routing table that stores neighbors whose keys match certain criteria, and keys stored in the routing table are exponentially distributed. Thus, a node knows more nodes in its logical vicinity than those whose keys are distant in the key space. Kademlia differs in its logical distance function: the logical distance between two DHT keys is defined as the exclusive-or of their bits. Figure 6.1 illustrates a Kademlia query.

As in other DHTs, in Kademlia, it is possible to trade-off logical progress for lower latency hops during the traversal from a query source to a target. Instead of selecting the node that makes the greatest progress toward the target in the key space, the source selects a hop that makes almost as much progress logically, but has a lower anticipated delay. Figure 6.2 portrays how I used network coordinates to optimize the DHT traversal process. First, I eliminate choices that do not make enough progress to the target. Choices that match in the same number of bits as the logically nearest key do not increase the expected number of

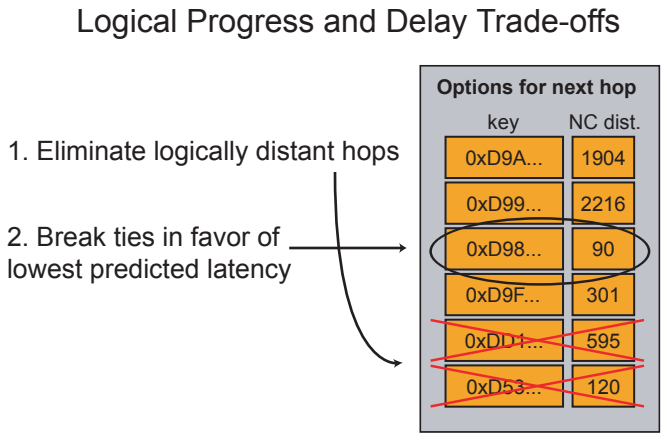

Figure 6.2: **Kademlia Traversal Trade-offs.** Instead of always selecting the hop that makes the most logical progress, a set of hops is available that makes roughly equivalent progress, creating a trade-off between logical progress and delay. First, to create this set, logically distant hops are eliminated; this retains hops that make similar progress to the target. Second, ties are broken in favor of lowest predicted latency, as predicted by the coordinates.

---

hops. Second, I use network coordinates to break logical ties in favor of the lowest predicted latency path. Eliminating logically distant choices still leaves a set of possibilities. With no additional measurement, network coordinates enable a good guess on which choice will be the lowest latency.

6.2.1 Distributed Hash Table Traversal Experiment

I modified an Azureus client so that it used network coordinates to optimize lookup delay. The experiment to evaluate the change in lookup delay first stored a set of keys in the DHT, then looked up each key using four distinct node selection methods, recording the time for the lookup operation. For each key, I ran the methods in random order. Each method selects one node from a small set, *i.e.,* is performing an anycast: all choices will make logical progress toward the target, some have lower latency than others. As illustrated in Figure 6.3, starting with the logically nearest known nodes to the target, XOR picks the logically nearest node, ORIGINAL picks the node whose latency as predicted by the original $2d + h$ coordinates is smallest, OPTIMIZED picks the lowest latency node as predicted by the optimized $4d + h$ coordinates, and RANDOM picks randomly from the set. Each node contacted returns its neighbors that are logically close to the target. This repeats until either a node storing the key is found or the lookup fails. Because Azureus performs DHT lookups iteratively, I was able to experiment with the lookup algorithm through code updates on only a single node.

I plot the distribution of delays from storing 250 keys and performing 2500 lookups in Figure 6.4. On average, I picked from among 5.5 potential hops. Compared to the XOR method, which always chooses the nearest logical node, the data show that OPTIMIZED reduces lookup delay by 33% at the 80^{th} percentile. It is 12% faster than the early version of the coordinates, ORIGINAL, also at the 80^{th} percentile. Two factors suggest these improvements are conservative. First, because no latency prediction information is currently returned to the caller, the optimization only affects the selection of the first hop. Second, I was not able to predict latencies to 34% of nodes due to version incompatibilities. I excluded lookups that timed out due

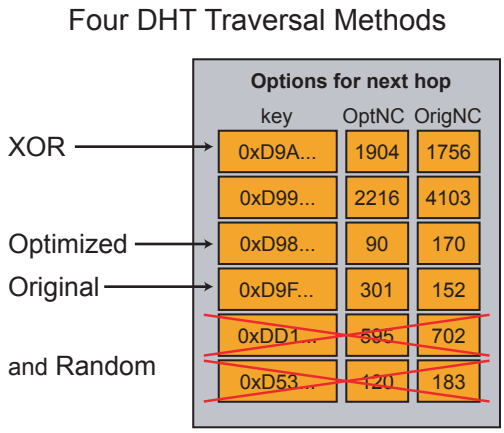

Figure 6.3: **Four DHT Traversal Methods.** Each traversal used a particular method to select the next hop to take from among the set of nodes that would make roughly equivalent progress: XOR selects the hop that makes the most logical progress; OPTIMIZED selects the hop that the new version of the coordinates predict to be of lowest latency; ORIGINAL uses the same method, but relies on coordinates formed without the techniques from Chapters 3 and 4; and RANDOM selects randomly from the sublist.

---

to dropped UDP messages to avoid dependence on a particular timeout handling mechanism. These data show that using network coordinates can provide a substantial application-level performance improvement and enable anycast decisions where a node is comparing its position to a small collection of possibilities in a real-world environment containing millions of nodes.

6.3 Locality-Biased Swarms in BitTorrent

In this section, I show how network coordinates can be used to optimize the set of overlay connections that constitute a wide-area file distribution mesh. The results show not only that network coordinates improve download times in the mesh, but, more broadly, that latency prediction can serve as a substitute for bandwidth prediction, for which there is currently no alternative method. The particular application is the BitTorrent swarm, where network coordinates help the central server, the *tracker*, make efficient predictions about which nodes in the swarm are near one another. This enables the nodes in the swarm to exchange data at higher rates and adheres to the *locality principle*, keeping network traffic with only local relevance local [22].

Recall from Section 4.2 that in BitTorrent an initial seeder divides a file that it wants to distribute into small pieces. The initial seeder then creates a metadata description of the file and sends this description to a per-file central server, called a tracker. Peers that want to download the file contact the tracker to bootstrap their knowledge of other peers and seeds. Trackers return a subset of nodes that have recently contacted them, creating a swarm exchange graph. Figure 6.3 illustrates the main steps a node takes when joining a swarm, extending the swarm exchange graph. Currently this graph is random. This section examines the benefit of a graph biased so that connections tend to be localized instead of random.

One of the main reasons for the popularity of the BitTorrent distribution protocol is that the provider of a file does not bear the expense of its distribution. This removes a major barrier for making

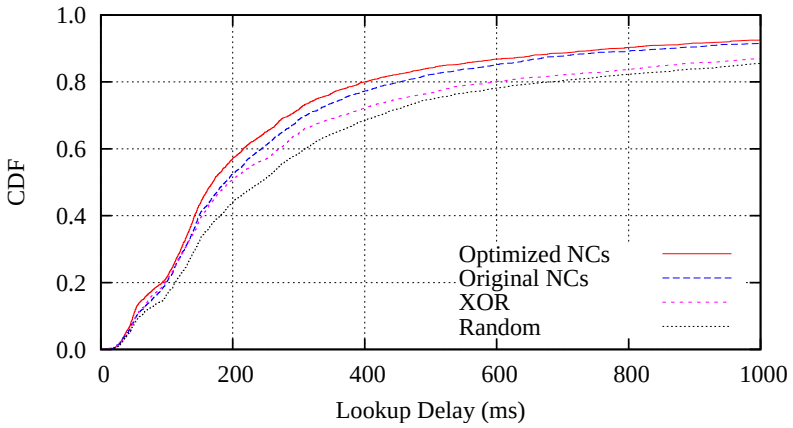

Figure 6.4: **DHT Traversal.** By choosing paths that are small detours in the logical space but lower latency, network coordinates improve lookup delay in Azureus's DHT.

---

content publicly available. Multicast trees also enable users to provide content without paying for upload bandwidth (*e.g.,* Scribe [13]). However, as nodes come and go, trees are challenging to maintain; meshes have less stringent requirements on which nodes can link to each other. Unlike a multicast tree, the BitTorrent protocol does not require any single participant to remain online, only that one copy of each piece be held by an online participant.

Unfortunately, random swarm exchange graphs cause two problems: excessive Internet Service Provider load and reduced download times for users. Because neighbors are randomly connected, many are likely to be positioned in a different ISP. In addition to causing long-distance traffic, these links tend to exhibit higher latency and lower bandwidth. Because nodes have only partial knowledge of the structure of the swarm, they can remain unaware of closer copies of the same pieces of the file. This results in many copies of the same piece passing through the same ISP gateway, creating an unnecessary, redundant load for which ISPs must pay. To limit this load, many ISPs have placed artificial capacity limits on their BitTorrent traffic. In turn, hampered users have attempted to circumvent these measures through encryption, alternative ports, and anonymization services. Unbiased swarms have slower downloads because of active ISP traffic shaping and the invisibility of nearby nodes in the swarm. The popularity of the protocol, together with its redundant traffic, has led to massive bandwidth consumption, with estimates ranging from 18 to 55 percent of Internet traffic, depending on measurement methodology [60].

While retaining the key benefits of the protocol, a simple change to the swarm exchange graph both improves users' download times and reduces inter-ISP bandwidth. As Bindal *et al.* suggest, making the swarm graph include edges to (mostly) local nodes instead of random ones reduces inter-ISP bandwidth without hurting client download times [8]. Figure 6.6 illustrates the change in the swarm graph from random to locally-biased.

Several methods could be used to implement locality-biased swarms. Bindal *et al.* suggest two methods: (1) performing deep packet inspection and modification at ISP gateways, editing the list of nodes coming from the tracker on-the-fly, and (2) building Internet topology maps into trackers so that they can create more efficient graphs. The former appears difficult to perform at line speed, especially because some of the traffic is encrypted and travels over many ports. The latter is more feasible, but the maps would need to be kept up-to-date with what is frequently proprietary information and could impose a memory constraint

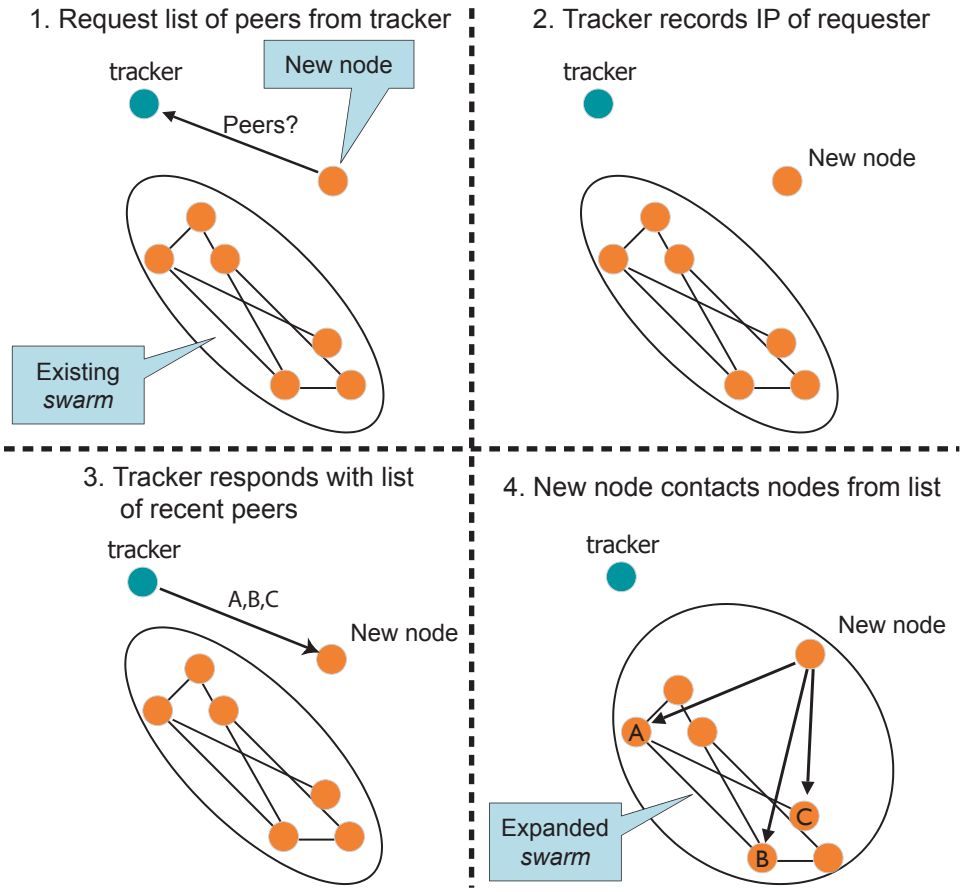

Figure 6.5: **Peer Discovery in BitTorrent.**

Recall from the overview of BitTorrent in Section 4.2 that a group of nodes currently exchanging pieces of a file is called a swarm.

1. A peer that wants to join a swarm and download a file first contacts the tracker for that file; trackers store the addresses of nodes in their swarm.

2. The tracker remembers the requester (via its IP address), so it can pass the requester along to future nodes that contact it.

3. The tracker responds with a list of peers for the new node to contact.

4. The new node contacts these nodes and enters the swarm.

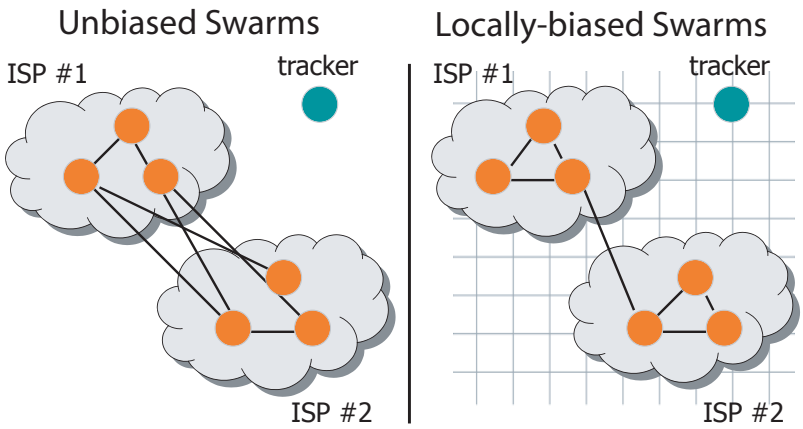

Figure 6.6: **Locality-biased Swarms.** Locality-biased swarms improve bandwidth for peers, due to less traffic shaping and higher bandwidth links, and reduce inter-ISP bandwidth.

---

on trackers.

I propose using network coordinates to provide an indicator of proximity. As Figure 6.3 shows, there are many possible nodes that the tracker could return. Instead of returning a random set, it can perform an anycast and return a locally-biased set. Instead of only storing addresses of nodes in the swarm, the tracker also records nodes' coordinates. The tracker can learn each node's coordinate as it contacts the tracker to request entry to the swarm. In Step 3 of peer discovery (Figure 6.3), instead of returning random recent peers, the tracker returns peers that are nearby the new node's coordinate. Using network coordinates, the tracker can approximate the relative proximity of groups of nodes that have never communicated.

Interestingly, the swarming application is not concerned with latency: while peers do send some short control messages to one another, inter-peer bandwidth is much more significant. Thus, the coordinates are approximating latency, which is acting as a proxy for bandwidth. Oppenheimer *et al.* and Lee *et al.* have investigated the inverse correlation between latency and bandwidth [47, 59]. Oppenheimer, in particular, found that latency can act as a proxy for bandwidth in many cases. This correlation is especially beneficial for overlay networks, because bandwidth measurement techniques, such as packet trains and active probing methods, have high overheads and error rates in practice [45]. It is not tractable for the tracker to obtain either $O(n^2)$ inter-node bandwidth or latency measurements. However, using network coordinates as a proxy, it can make relative estimates of inter-node bandwidth.

6.3.1 Locality-Biased Swarm Experiment

To evaluate the effect of locality-biased swarms, I hosted a tracker on a machine at my university and experimented with the graph created by the sets of nodes it returned. The Azureus developers had previously modified the source that the clients run so that they would couple their network coordinate along with each tracker request.

I modified the tracker so that it biased the links in the swarm exchange graph according to four different neighbor bias models. RANDOM randomly links nodes in the swarm exchange graph; this is the current model for BitTorrent. NEAREST COORDINATE links each node to its nearest neighbors in Azureus's coordinate space, which now uses the techniques from Chapter 4. LOWEST LATENCY shows the effect of

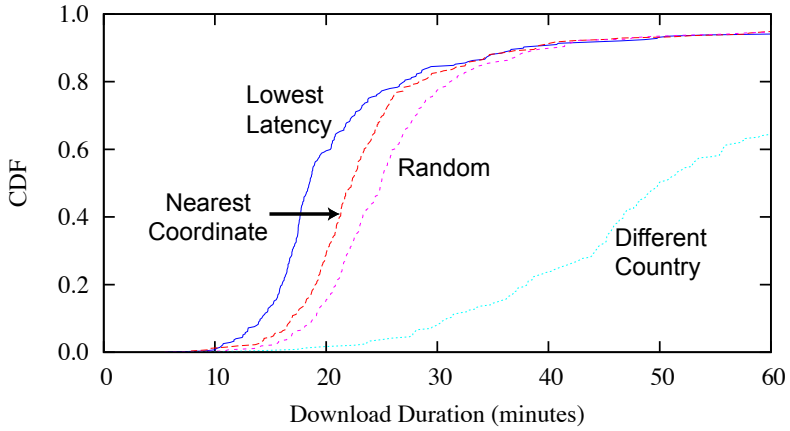

Figure 6.7: **Faster Downloads with Locality-Biased Swarms.** When nodes discover exchange partners that are nearby, client-perceived download times reduced in our experiment on PlanetLab.

---

directly proxying latency for bandwidth. It captures the hypothetical situation that would occur *if* perfect information on latency to all nodes were available at the tracker. This bias shows how well true latency information acts as a substitute for bandwidth. It was possible to create this bias because the swarm is small enough to take the inter-node latency of all n^2 pairs. Lastly, DIFFERENT COUNTRY acts as a lower bound by forcing the graph to have long latency edges.

In order to ensure a fair comparison, the same nodes were linked to the seed in each experiment, all nodes received the same number of neighbors (four), and every node formed a link to a node in a different autonomous system to eliminate the possibility of partitions. To conduct the swarm localization experiment, I installed Azureus clients on 328 PlanetLab nodes. All nodes started the download process at approximately the same time. The file they downloaded was 180 megabytes in size and was seeded by the tracker.

The data confirm my expectations on download speed, provide evidence in support of the heuristic that latency and coordinate distance can act as a proxy for bandwidth, and show that locality-biased swarms reduce overall network usage. I plot the distribution of the download duration for each neighbor bias model in Figure 6.7. The data show that LOWEST LATENCY, representing access to a hypothetical, complete latency matrix, had 26% faster median downloads than RANDOM, reinforcing the latency-bandwidth heuristic. Azureus' coordinates do not provide a perfect approximation of this matrix and are not capable of realizing all of this improvement: it is 11% faster than RANDOM at the median. Table 6.1 shows the mean network usage of each piece using the different biases. Network usage is calculated as the latency-bandwidth product and has a deliterious effect on congestion control algorithms when large, in addition to simply consuming resources [40, 65]. The data show that, because each piece travels a shorter distance in the network with the two locality-biased methods, they consume fewer routing and ISP resources. Lastly, I plot the correlation between latency and bandwidth and between coordinate distance and bandwidth in Figure 6.8. The data show that there is a fairly strong correlation in both cases and that the coordinate distance correlation is only slightly weaker than that for latency. Locality-biased swarms both improved download times in the mesh and reduced network resources that BitTorrent swarms consume.

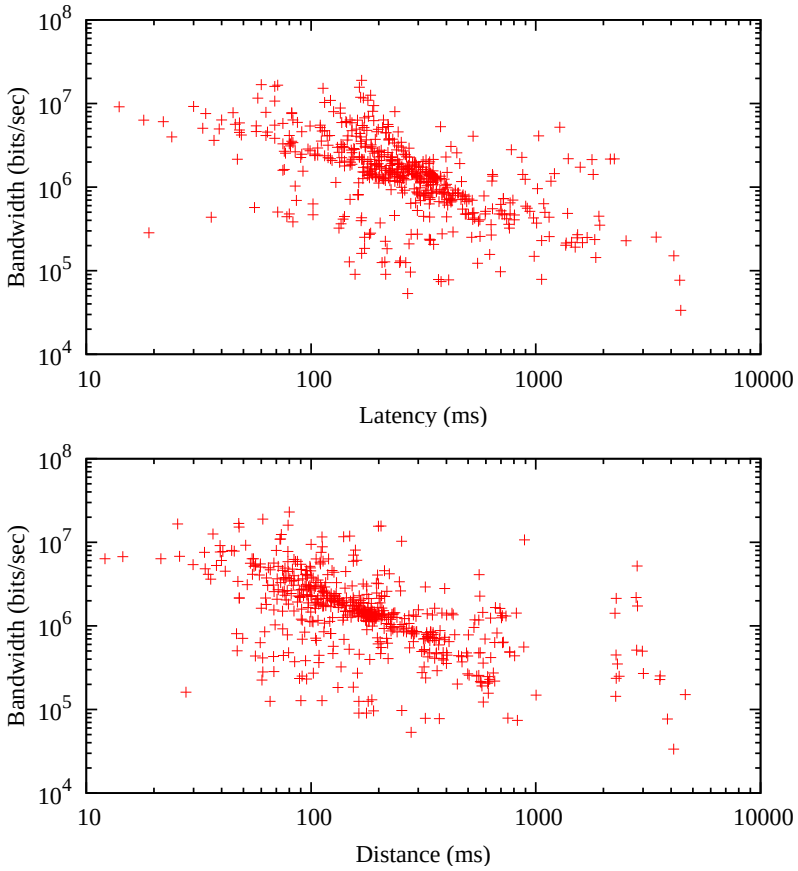

Figure 6.8: **Correlations with Bandwidth.** Within Azureus's swarm, both latency and coordinate distance act as predictors of bandwidth. The grouping and slope of the cluster of points in the log-log plots shows a power-law correlation among the data. Both latency and coordinate distance show a moderately strong correlation to bandwidth, with an r^2 of -0.58 and -0.54 respectively. This is in keeping with the latency-bandwidth correlation Oppenheimer *et al.* found: an r^2 of -0.59 [59].

Bias	Network Usage
Lowest Latency	248.2 (-17%)
Nearest Coordinate	262.5 (-12%)
Random	297.6 (0%)
Different Country	354.8 ($+19\%$)

Table 6.1: **Network Usage of each bias.** Network usage, computed as the bandwidth-delay product, captures the amount of data in transit in a network. Both locality-aware biases, LOWEST LATENCY and NEAREST COORDINATE, reduce the data in transit. This is significant because BitTorrent constitutes a large portion of total Internet traffic.

6.4 When are Network Coordinates the Appropriate Tool?

Network coordinates are not a panacea. They are well-suited to certain types of applications and not others. Even in the applications for which they are well-suited, they can exhibit high prediction error depending on the network. In this section, I discriminate between what are and are not useful applications for this new "hammer" in the distributed application developer's toolbox.

Some criticisms of network coordinates are well-founded and some stem from a misunderstanding — typically an overstatement — of their utility. Wong *et al.* make several criticisms that were well-founded at the time of writing [79]. They argue that the embedding process necessarily introduces errors on realistic networks, that properly tuning implementation parameters is non-trivial, and that coordinates need continuous re-computation. These criticisms are valid, but much work, including techniques in this thesis, addresses them to a great extent. Using or promoting network coordinates in inappropriate contexts is more divisive. This overstatement of network coordinates' utility takes two forms: arguing that they easily generalize from latency to other characteristics and depicting them as a complete resource discovery framework.

Several pieces of research have overstated the generalization of network embeddings from latency to other network and node characteristics. One example is arguing that measurements, such as load, can be combined with latency into a meaningful metric space. Co-authors and I succumbed to this fallacy when we attempted to add arbitrary metrics into a *cost space* [73]. The metric we added was load: latency made up the first three dimensions of a node's coordinate and its load was the fourth. We then redefined distance as the Euclidean distance of the latency components plus some factor of load. One problem with adding characteristics such as load into a multidimensional space is that all queries using the same coordinate system need to declare the same trade-off between latency and load. In many applications, however, queries that want to find a node near a location in the coordinate space will want to apply different weights to the load component. A second problem is that, unlike latency, load is not *transitive*: the load of one machine has little bearing on the load of another. Thus, a node has no way to adjust its own load dimension with respect to its neighbor's as it can with latency. It is possible to include characteristics such as load as separate scalar components that describe a particular node, but these characteristics can have no "push" or "pull" as they have no bearing on the rest of the network. Resource measurements that include characteristics orthogonal to latency, including load and the services a node is running (or capable of running), should not be added to a *cost space*. Arguing for this has exaggerated the power of network coordinates.

Researchers have also overstated network coordinate utility by claiming that other network coordinates can be embedded with low error. Network coordinates are often described as a general method for embedding and predicting *some* characteristic. However, is appears infeasible to directly embed network characteristics other than latency. In order to extract coordinates of reasonable accuracy from an input matrix, that matrix cannot contain many large violations of the triangle inequality. Inter-node latencies experience these violations to a degree that depends on the particular network, as discussed in Section 4.3.3. Inter-node bandwidths, in particular, experience so many violations of the triangle inequality that a low error embedding appears impossible. For example, let node a have a bandwidth of one megabyte per second to both nodes b and c. Any bandwidth between b and c greater than two megabytes per second will be a violation of the triangle inequality. But the bandwidth between b and c could be essentially any value: it could be orders-of-magnitude larger than a's bandwidth to them. Figure 6.9 portrays the extent of triangle inequality violations for latency and bandwidth of PlanetLab nodes. The figure shows cumulative distributions of the fraction of triangle inequality violations for each node with respect to all other node pairs. The data portray that the percentage of violations is far higher with the bandwidth dataset, suggesting that an embedding process simply would not result in meaningful coordinates.

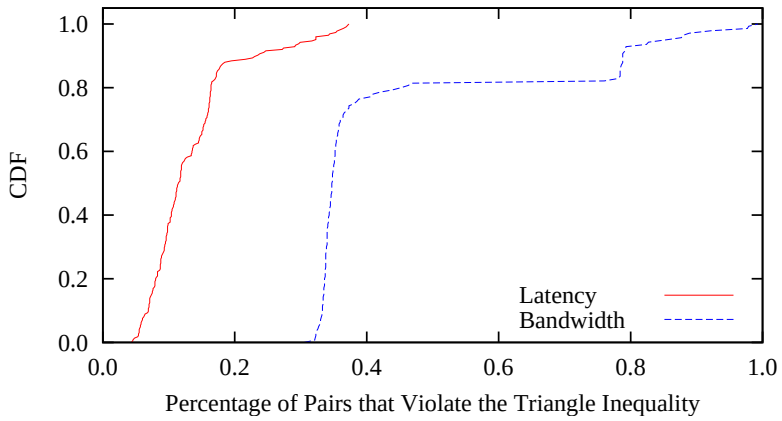

Figure 6.9: **Latency and Bandwidth Triangle Inequality Violations.** The figure shows the percentage of triangle inequality violations for each node to all other pairs of nodes. Collected by Lee *et al.*, the bandwidth data are from a matrix of 141 PlanetLab nodes [47]. I collected the latency matrix of 226 PlanetLab nodes as part of earlier work [46].

The first fallacy surrounding network coordinates was that characteristics other than latency are easily embeddable. The second stems from the desire to depict them as a complete resource discovery framework. A revealing example of inappropriate use is a comparison by Freedman *et al.* between their system, OASIS, and network coordinates [28]. OASIS is a long-lived overlay infrastructure that answers questions of the form: "Which server should the client contact for nearby access to a particular service?" Freedman *et al.* assume that many instances of a possible service exist and that clients want to find one instance that will service them for an extended period. OASIS aggregates load, liveness, and position information into a replicated database. Because it acquires significantly more information than network coordinates provide, it makes more sensible decisions at higher cost. OASIS strikes a different balance in the cost of acquiring an answer and the quality of that answer. Network coordinates provide position — and only position — information at the cost of no additional messaging, assuming on-going application traffic is used for coordinate maintenance. Because they do not include liveness, load, or other metrics, network coordinates remain mute when a resource decision extends beyond latency. Network coordinates could certainly be a *component* for a resource discovery system. For example, they could feed latency information into OASIS or could route to locations in a network to start detailed investigations of a local area (see Section 5.6). Because network coordinates are not a complete resource discovery framework, however, it makes little sense to compare them to one.

In this chapter, I showed two applications where network coordinates did result in a significant performance improvement. Using network coordinates in these contexts is more practical than using more comprehensive, heavyweight resource discovery tools. In the DHT traversal application, the expense of determining which hop is truly fastest (*i.e.,* pinging all of them) is not worthwhile because then each traversal would be the sum of the maximum delays. Instead, making quick, typically accurate predictions improves aggregate application response time. In the locality-biased swarm application, the tracker can be bombarded with requests hundreds or thousands of times per second: it does not have time to perform an extensive evaluation of which nodes should connect to each other. Providing hints to swarm exchange graph formation outweighs the cost of forming an optimal graph. From this, I conclude that using network coordinates makes

sense when a quick, typically accurate guess outweighs an extensive search for the optimal choice.

Candidates for network coordinate applications are limited since latency is the only network characteristic that can be embedded with reasonable prediction accuracy. Latency prediction includes a broad range of applications — even acting as a substitute for bandwidth prediction in the swarm application — but this range is bounded. Intuitively, network coordinates provide a scalable method for approximating all-to-all latency matrices and a spatial framework for understanding where users, data, and services are positioned in a network. They can predict the relative proximities of sets of nodes, as they did in the swarm application, and they can determine which servers lie near the "line" between two endpoints. They can also be used as a routing substrate, on which higher-level services such as multicast and remote service discovery can be built.

6.5 Conclusion

In this chapter, I focused on applying network coordinates to two applications, DHT traversal and locality-biased swarms, and drew conclusions on when network coordinates are the appropriate tool. The experiments showed that the coordinates provided a significant application-level boost through readily-accessible locality predictions. In the DHT traversal application, they helped trim lookup delay by 33% compared to the most direct logical path. In the locality-biased swarm application, they diminished network usage by 12% and improved download times by 11%.

Several new applications are beginning to incorporate ideas similar to those in this chapter. For example, RedSwoosh is a web browser plug-in that acts like a web cache [68]. Instead of residing at an ISP (as in a traditional cache), the plug-in actively queries other *local* instances of the plug-in, attempting to download content from these peers first. Other overlays that focus on just-in-time video streams, such as Tribler [66] and GridCast [33], are starting to use similar locality techniques, including network coordinates, to make quick, accurate estimates of inter-node locality.

Chapter 7

Related Work

Because network embeddings are a young field, combining aspects of systems, networking, and theory, little prior work has focused directly on the main themes of this thesis: measuring, refining, and using live coordinate systems. Each chapter, however, has strong connections to prior and current research. In this chapter, I discuss other methods for evaluating coordinate systems, a different proposal for stabilizing coordinates, methods for routing that could be applied to network coordinates, and recent research on wide-area resource discovery.

7.1 Evaluation Metrics

I proposed a new metric for measuring coordinate systems in Chapter 3. My metric measures *stability*, or coordinate change over time. Three pieces of research have proposed other metrics for evaluation. These metrics can be used to compare across embedding methods (*e.g.,* with or without height; hyperspherical coordinates) or across different latency distributions (*e.g.,* PlanetLab and Azureus).

Lua *et al.* propose metrics that try to capture application impact more broadly than relative error [49]. Their *relative rank loss* (RRL) determines how well an embedding respects the relative ordering of all pairs of neighbors. Thus, for each node x,

$$\text{if } (d_{xi} > d_{xj} \wedge l_{xi} < l_{xj}) \text{ or } (d_{xi} < d_{xj} \wedge l_{xi} > l_{xj}),$$

then the distances d_{xi}, d_{xj} between coordinates lead to an incorrect prediction of the relative latencies l_{xi}, l_{xj} (*i.e.,* they are incorrectly ranked). This metric is important for applications that make decisions dependent on the relative ordering of nodes, for example, when deciding which node to include in a routing table slot. Although RRL is a valuable extension over relative error, it disproportionally weighs nearby nodes, which are often ranked incorrectly with negligible application impact.

Co-authors and I extended RRL to capture the *magnitude* of each incorrectly ranked pair [64]. This is based on the intuition that, in many cases, the damage done by 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. In this same work, we also proposed *relative application latency penalty* (RALP), which expresses the percentage of additional latency that an application will observe when using network coordinates for rank ordering. This metric is generated by summing the relative penalty l_{ij}/l_{xi} for all pairs that are incorrectly ranked. These metrics provide a useful hint at application performance; in Chapter 6 I measured the actual performance of two applications.

Zhang *et al.* take a different approach to examining the accuracy of a coordinate assignment [80]. Instead of bundling all nodes together, they partition nodes based on latency and measure the accuracy within a given range. This metric, *range accuracy*, measures the prediction accuracy (*i.e.,* relative error) of coordinates within a given range r. Range accuracy is useful because distributions of relative error can overstate the importance of low latency nodes: for low latency pairs, even a small absolute prediction error results in a large relative error.

7.2 Stabilization

The application-level coordinates I discuss in Section 3.5 increase coordinate stability without damaging accuracy. In other work that attempts to stabilize coordinate systems, de Launois *et al.* modify Vivaldi to prevent oscillations in the presence of triangle inequalities [21]. Their work, called *SVivaldi*, introduces 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. Like early work on network coordinates that depicts assigning coordinates as a one-time operation, SVivaldi locks nodes' positions in place until some threshold of inaccuracy is breached. This is in contrast to my methods for stabilization, which allow coordinates to smoothly evolve over time.

7.3 Coordinate-Based Routing

In Chapter 5, I examined constructing a locality-aware routing substrate on top of a network coordinate system. I started with $\hat{\theta}$-routing, an algorithm designed for efficient routing in a static two-dimensional Euclidean space. I then developed several techniques that enabled the algorithm to function in realistic, decentralized environments and examined its performance on real-world networks. Several other pieces of research have examined coordinate-based routing, although none has attempted to build a general-purpose substrate. I highlight the current state-of-the-art: Tulip, Mithos, and Compact Routing.

Tulip

Abraham *et al.* designed *Tulip*, which routes to target nodes without a virtual coordinate system [1]. Each node selects a random "color" out of $O(log\ n)$ choices and maintains a *color list* which contains all nodes of its color. It also maintains a *vicinity list*, containing the nearest node of every other color. A node routes a message by sending it to the node in the vicinity list of the target's color. That node routes the message directly to the target using its color list: Tulip's two-hop routing is locality-aware. Tulip uses gossip and direct node-to-node measurements to place neighbors into its vicinity list and to ensure the liveness of its color list. These measurements result in a high maintenance overhead. Because the query source must directly specify the target (*e.g.,* color=blue, id=57), it is unclear how Tulip could be extended to support closest node queries or queries to an arbitrary location.

Mithos

The work closest to what I present in Chapter 5 is Waldvogel and Rinaldi's *Mithos*, a locality-aware routing substrate on top of virtually-assigned coordinates [78]. Each node calculates a static coordinate based on its network location and maintains links to its immediate neighbors in each quadrant.

Interestingly, it can be expressed as θ-routing ($\hat{\theta}$-routing without the rings) with $\theta = \pi/2$. Like θ- and $\hat{\theta}$-routing, messages are routed greedily towards a target.

Mithos has three shortcomings. First, since routing tables do not contain long-distance links, hop count is linear unless the coordinates are of high dimensionality. Second, because these nearby links are also used to establish coordinates and because others and I have established that purely local neighbors lead to globally inaccurate coordinates (see Section 4.4.1), Mithos' coordinates are likely to be poor quality. Lastly, Mithos assigns coordinates in a one-time operation, resulting in periods of poor accuracy prior to each reassignment.

Compact Routing

Building on Hassin and Peleg's $\hat{\theta}$-routing, Abraham and Malkhi designed Compact Routing [2]. Compact Routing is significantly more complicated than $\hat{\theta}$-routing, but reduces per-node state to near optimal. The key difference between the two is the diameter of the rings that surround each node. In both, the diameters of rings grow exponentially. In Hassin and Peleg's work, all nodes' k rings are the same set of sizes: $b^1, b^2, \ldots, b^{k-1}$. In Compact Routing, the base of the ring size varies: the radii of node i's rings might be ${b_i}^1, {b_i}^2, \ldots, {b_i}^{k-1}$, while node j's are ${b_j}^1, {b_j}^2, \ldots, {b_j}^{k-1}$. These differently-sized rings mean that some nodes have much further reach than others, while others have detailed views of their local area. The routing strategy takes advantage of this by forwarding the message to the nearest node whose rings are large enough to get the message into the vicinity of the target in one hop. From there, nodes with progressively smaller sets of rings are used to zoom in on the target node. As with $\hat{\theta}$-routing, Compact Routing needs to be altered to find the closest node to an abstract target point.

For implementation purposes, $\hat{\theta}$-routing made sense. It has a much simpler construction because all nodes' rings are the same size and routing proceeds by greedily minimizing distance to the target. $\hat{\theta}$-routing has the same number of expected hops (logarithmic), but has more state. In fact, Abraham and Malkhi recommended we use $\hat{\theta}$-routing over their work.

7.4 Locality-Aware Anycast

I used network coordinates to improve resource allocation decisions in Chapter 6. In the first application, distributed hash table (DHT) traversal, nodes use network coordinates to quickly select the closest node to themselves from a set without performing any extra measurement. Any hop made logical progress but coordinates allowed prediction of which choice leads to a low latency path. Dabek *et al.* perform a similar optimization in Chord by selecting which nodes should be members of the routing table based on coordinate distance [20]. In the second application in Chapter 6, swarm localization, a centralized server in BitTorrent uses network coordinates to determine the relative proximity of large collections of nodes, creating a more efficient mesh than BitTorrent's current random graph. As noted in Section 6.3, Bindal *et al.* simulated the effects of locality-based swarms on a simplified model of the Internet [8]. Their findings were similar, however, they did not include a robust method for forming the biased swarm exchange graph.

7.4.1 Meridian and OASIS

Network coordinate implementations focus on maintaining a generalized latency prediction infrastructure that works well for most queries. A separate body of work has focused more directly on the problem of finding the nearest and "best" of many replicated services. The primary examples of this work are Meridian and OASIS.

Meridian finds the nearest overlay node (*i.e.,* one running Meridian) to an arbitrary non-Meridian node in the Internet through a set of pings that progress logarithmically closer to the target. Before queries can occur, each Meridian node places its neighbors into rings. Like $\hat{\theta}$-routing, the diameters of rings grow exponentially and all nodes' k rings are the same set of sizes. Starting at a Meridian node, queries hop closer to the target. At each step, the current hop requests all of its neighbors in the target's ring and half of the nodes in the adjacent rings to probe the target. The current hop forwards the request to the neighbor reporting the lowest latency to the target. This neighbor then repeats this process recursively, exponentially minimizing latency at each step.

More recently, Freedman *et al.* developed another reactive anycast mechanism: OASIS [28]. OASIS is explicitly designed to help clients find a "good" server out of many — not just the closest. OASIS uses Meridian as its locality component and requires a reliable core of hosts to map clients to nearby servers, which are assumed to be long-lived. OASIS also adds other attributes, liveness and load, to make more precise queries possible.

In Meridian and in OASIS, each query has a large measurement overhead. Also, results may be inaccurate when a node's ring membership changes due to churn. If used to solve Azureus' swarm localization problem, all active peers in a swarm would form a Meridian overlay and each peer would initiate a query for itself. Using the parameters from Wong *et al.*, each lookup could entail approximately 256 messages, assuming two thousand peers in a swarm. Meridian would also require modification to find the k-closest nodes, not simply the closest node. In contrast, using network coordinates to find the closest peer requires only internal coordinate comparisons by the tracker.

As discussed in Section 6.4, however, quantitative comparisons can obscure the distinct purposes of network coordinates and these anycast services. Meridian and OASIS are designed for the case where contact with the service will be frequent and long-lived enough to outweigh the high upfront cost of finding the best service. With their current levels of accuracy (good but not perfect) and maintenance (zero), network coordinates provide low-cost, typically accurate latency information to applications where finding the exact answer is not worthwhile, but repeatedly finding a good answer leads to aggregate savings.

Chapter 8

Conclusions

8.1 Lessons Learned

This thesis stemmed from my research group's work on Stream-Based Overlay Networks [65]. My task as part of that project was to create an accurate and stable network coordinate system on PlanetLab. I found this so engaging that I could hardly wait for the main SBON paper to be finished so that I could focus on the details of the coordinates themselves. The professors involved in the project, Margo Seltzer, Matt Welsh, and Mema Roussopoulos, all said that "dozens" of thesis topics lay in this project. I take from this that pursuing an interesting project (as long as one has done sufficient background work to make sure the research is novel) can easily lead to where one would never have expected it.

One place where I certainly never expected my work to end up was within a real industrial project used by millions of people. I had read about Azureus's use of network coordinates online, but had not considered that they would be interested in a collaboration. After publishing the WORLDS paper [64], Mike Parker put me in touch with Paul Gardener of Azureus and the collaboration began. Initially I thought working with Azureus was going to be a waste of time and a diversion from research. But because he understood Azureus's clients massive code base, Paul was a great help in setting up the tools I needed to build and test changes to Azureus's "wild" network coordinates. Being driven by this real-world application was fantastic and provided incontrovertible evidence that network coordinates could provide reasonable latency prediction and improve application-level performance in an Internet-scale network. I was exceptionally lucky to have formed this collaboration. The lesson I draw from this is that industrial collaboration, perhaps especially in Systems, can provide both scope and direction on what are good problems to solve and confidence that one's solutions apply to realistic contexts. A second lesson is that collaborations can yield data impossible to gather in any other way.

A final lesson is that having a peer researcher whom you respect and trust is incredibly beneficial for creating new, exciting, and well-considered research. At Wisconsin, I had a great partner in Matt McCormick as we constructed a file system designed for speedy web object retrieval. My work with Dan Ellard was equally good in more of a mentoring sense: he was the *senpai* and showed how to run a successful group project. My work with Peter Pietzuch was also an excellent collaboration, one which we have continued since his departure for Imperial College London. Establishing relationships like these is something I will strive for in the future.

8.2 Future Directions

My thesis leaves open several distinct directions for future research.

- Both results from the two applications in Chapter 6 were promising and I would like to integrate them into the Azureus main source tree. In particular, I would like to measure the effect of locality-biased swarms at a more "macro" level. A good way measure this change would be to watch aggregate BitTorrent traffic at ISP gateways and observe the decrease in traffic when locality-biased swarms are put into effect. Or it may be possible to perform a more fine-grained experiment, similar to the one I ran in Chapter 6, where a popular tracker uses several different methods to create the swarm exchange graph. Even if this measurement is not possible, I would still like to work with the Azureus developers to integrate the work into their trackers. Their content is dominated by streaming video and they are actively working on minimizing the load on their servers without increasing stream latency. It seems that the applications from this chapter could have a positive effect on ISPs, Azureus (as a content provider), and end users.

- While the latency filters discussed in Chapter 3 worked well in practice, more in-depth analysis of wide-area inter-node latency data would yield interesting results. One track this analysis could take is in developing automatic methods to tune the filter's two variables, its size and percentile. Alternatively, a different filter, perhaps drawn from *robust statistics*, could place the empirical results on firmer theoretical ground. Another track would be to see if streams of latency measurements could reveal more information to applications than just latency. Certainly the inference of bandwidth from latency in Section 6.3 is an example of this. Other work could directly measure link reliability and then perhaps expand the coordinates to encompass this information, much like *confidence* encompasses the reliability of each coordinate's latency estimate. One example of this would be to generate an "error bar" for latency prediction: a range within which the true (current) latency is likely to be. It also may be possible to incorporate additional, already extant information, such as each endpoint's autonomous system, to form better estimates.

- As I noted in the beginning of Chapter 5, Peter Pietzuch's research group at Imperial is actively working on an decentralized implementation of $\hat{\theta}$-routing. His group is also pursuing the theoretical question that was left unresolved: how to guarantee, in a live system with changing coordinates, that the closest node to a target is found. I also said that routing is a building-block for higher level components, such as multicast and remote service discovery. I would like to build (or see built) these components.

  Even these components are still fairly rudimentary. As co-authors and I argued in previous work, it may be possible to apply numerous algorithms from computational geometry, such as minimum spanning trees, resource replication, and cluster analysis, to collections of network coordinates [63]. One simple example of this is a web cache that is placed at the centroid of a set of web clients that wanted access to the same data. Similarly, a server hosting a distributed game could be positioned the centroid of the players' coordinates, leading to fair access times.

8.3 Final Thoughts

This thesis focused on translating the successes of network coordinates in the lab to more realistic scenarios, measuring these scenarios, and improving on them where possible.

I examined two overarching aspects of live network coordinate systems. Chapters 3 and 4 focused on constructing accurate coordinate systems on live networks. I introduced three techniques, latency filters, update filters, and neighbor decay, that each addressed a distinct problem that occurred in live systems. These problems only appeared as the research community was attempting to translate simulation successes with network coordinates to more practical, live implementations, including my group's Stream-Based Overlay Network. I evaluated these techniques on PlanetLab and within the Azureus BitTorrent network. This evaluation on Azureus, which contains more than a million nodes at any given time, provided particularly realistic data due to its size, rate of churn, and widespread, public use. In addition to showing the viability of the techniques, the evaluation showed that using network coordinates within huge applications is practical.

The second half of the thesis focused on how network coordinates can assist applications with locality-aware routing and resource selection. In Chapter 5, I built on previous theoretical work on routing within a system of static, two-dimensional coordinates. I generalized this work to live, multidimensional network coordinate systems. I also evaluated its performance in simulation with coordinates derived from a real network. Other researchers are in the process of taking this work and adding it to the Pyxida project, the open source implementation of network coordinate research that fellow researchers and I have done over the past two years. Chapter 6 examined the application impact of network coordinates on two anycast decisions, DHT traversal and locality-biased swarms in BitTorrent. For all the criticism network coordinates have received, I provided concrete evidence that they are sufficiently accurate for particular applications. One conclusion that I drew from the success of these applications is that network coordinates are the right tool when instant access to a good locality estimate is better than an expensive query for the "best" choice.

Prior to the growth of the Internet, understanding locality was, in many senses, a different problem. Locality problems tended to be local, on the order of buses, remote memory, and machine rooms. Locality is now a problem whose answers can only be approximated at best. Network coordinates are one method for approximating the holistic picture that was previously available in local area environments. With a group of participants' coordinates in hand, a node has a new viewpoint from which it can understand the shape and dynamics of its network. This viewpoint is fundamentally different from the one provided by a collection of DHT or IP addresses. As wide-area distributed applications, from Internet-based telephony to television, become ubiquitous, developing locality-aware approaches to distributed systems has become essential to both users and providers. As end users and providers become more mobile in the future, new challenges in locality research will continue to emerge.

Bibliography

[1] Ittai Abraham, Ankur Badola, Danny Bickson, Dahlia Malkhi, Sharad Maloo, and Saar Ron, "Practical Locality-Awareness for Large Scale Information Sharing," in *Proc. of International Workshop on Peer-to-Peer Systems*, (Ithaca, NY), Feb. 2005.

[2] Ittai Abraham and Dahlia Malkhi, "Compact Routing on Euclidian Metrics," in *Proc. of Symposium on Principles of Distributed Computing*, (St. John's, Newfoundland, Canada), July 2004.

[3] Akamai, Inc. `http://www.akamai.com/`.

[4] Réka Albert, Hawoong Jeong, and Albert-László Barabási, "The diameter of the world wide web," *Computing Research Repository*, volume cond-mat/9907038, 1999.

[5] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris, "Resilient Overlay Networks," in *Proc. of the Symposium on Operating Systems Principles*, (Banff, Alberta, Canada), Oct. 2001.

[6] "Azureus BitTorrent Client." `http://azureus.sourceforce.net/`.

[7] Yair Bartal, Nathan Linial, Manor Mendel, and Assaf Naor, "On metric ramsey-type phenomena," in *Proc. of Symposium on Theory of Computing*, (San Diego, CA), 2003.

[8] Ruchir Bindal, Pei Cao, William Chan, Jan Medved, George Suwala, Tony Bates, and Amy Zhang, "Improving Traffic Locality in BitTorrent via Biased Neighbor Selection," in *Proc. of International Conference on Distributed Computing Systems*, (Lisbon, Portugal), July 2006.

[9] Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia, "Routing with Guaranteed Delivery in Ad Hoc Wireless Networks," *Journal of Wireless Networks*, volume 7, number 6, 2001.

[10] Edouard Bugnion, Scott Devine, and Mendel Rosenblum, "DISCO: Running Commodity Operating Systems on Scalable Multiprocessors," in *Proc. of the Symposium on Operating System Principles*, (St. Malo, France), 1997.

[11] Fabian Bustamante and Yi Qiao, "Friendships that last: Peer lifespan and its role in P2P protocols," in *Proc. of International Workshop on Web Content Caching and Distribution*, (Hawthorne, NY), October 2003.

[12] Arthur R. Butz, "Alternative Algorithm for Hilbert's Space-Filling Curve," *IEEE Transactions on Computers*, pages 424–426, April 1971.

[13] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron, "Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure," *IEEE Journal on Selected Areas in Communication*, volume 20, Oct. 2002.

[14] Yan Chen, Khian Hao Lim, Randy H. Katz, and Chris Overton, "On the Stability of Network Distance Estimation," *SIGMETRICS Performance Evaluation Review*, volume 30, number 2, 2002.

[15] Bram Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. of Workshop on Economics of Peer-to-Peer Systems*, (Berkeley, CA), June 2003.

[16] Manuel Costa, Miguel Castro, Antony Rowstron, and Peter Key, "PIC: Practical Internet Coordinates for Distance Estimation," in *Proc. of International Conference on Distributed Computing Systems*, (Tokyo, Japan), March 2004.

[17] Landon Cox and Brian Noble, "Samsara: honor among thieves in peer-to-peer storage," in *Proc. of the Symposium on Operating Systems Principles*, (Bolton Landing, NY), Oct. 2003.

[18] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris, "Practical distributed network coordinates," in *Proc. of Workshop on Hot Topics in Networks*, (Cambridge, MA), November 2003.

[19] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris, "Vivaldi: A Decentralized Network Coordinate System," in *Proc. of SIGCOMM*, (Portland, OR), Aug. 2004.

[20] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, Frans Kaashoek, and Robert Morris, "Designing a DHT for Low Latency and High Throughput," in *Proc. of the Symposium on Networked Systems Design and Implementation*, (San Francisco, CA), March 2004.

[21] Cedric de Launois, Steve Uhlig, and Olivier Bonaventure, "A Stable and Distributed Network Coordinate System," technical report, Universite Catholique de Louvain, December 2004.

[22] Peter Denning, "The Locality Principle," *Communications of the ACM*, volume 48, July 2005.

[23] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl, "Globally Distributed Content Delivery," *IEEE Internet Computing*, volume 6, September 2002.

[24] "eMule File Sharing Client." `http://emule-project.net/`.

[25] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos, "On Power-law Relationships of the Internet Topology," in *Proc. of SIGCOMM*, (Cambridge, MA), 1999.

[26] Gregory Finn, "Routing and Addressing Problems in Large Metropolitan-scale Internetworks.," Technical Report RR-87-180, University of Southern California, Los Angeles, CA, 1987.

[27] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang, "IDMaps: a Global Internet Host Distance Estimation Service," *IEEE/ACM Transactions on Networking*, volume 9, number 5, 2001.

[28] Michael Freedman, Karthik Lakshminarayanan, and David Mazières, "OASIS: Anycast for Any Service," in *Proc. of the Symposium on Networked Systems Design and Implementation*, (San Jose, CA), May 2006.

[29] Krishna Gummadi, Stefan Saroiu, and Steven Gribble, "King: Estimating Latency between Arbitrary Internet End Hosts," in *Proc. of Internet Measurement Workshop*, (Marseille, France), Nov. 2002.

[30] James Gwertzman and Margo Seltzer, "World-Wide Web Cache Consistency," in *Proc. of USENIX Technical Conference*, (San Diego, CA), 1996.

[31] Yehuda Hassin and David Peleg, "Sparse Communication Networks and Efficient Routing in the Plane," in *Proc. of Symposium on Principles of Distributed Computing*, (Portland, Oregon), 2000.

[32] Hugues Hoppe, *Surface Reconstruction from Unorganized Points*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

[33] Cheng Huang, Jin Li, and Keith Ross, "Peer-Assisted VoD: Making Internet Video Distribution Cheap," in *Proc. of International Workshop on Peer-to-Peer Systems*, (Bellevue, WA), Feb. 2007.

[34] Bradley Huffaker, Marina Fomenkov, David Moore, and kc claffy, "Macroscopic analyses of the infrastructure: measurement and visualization of Internet connectivity and performance," in *Passive and Active Network Measurement Workshop*, (Adelaide, Australia), Mar. 2000.

[35] Stephan Husen, Ralph Taylor, Robert Smith, and Henry Healser, "Changes in geyser behavior and remotely triggered seismicity in Yellowstone National Park produced by the 2002 M7.9 Denali fault earthquake," *Geology*, volume 32, June 2004.

[36] John Jannotti, David Gifford, Kirk Johnson, Frans Kaashoek, and James O'Toole, Jr., "Overcast: Reliable Multicasting with an Overlay Network," in *Proc. of Symposium on Operating Systems Design and Implementation*, (San Diego, CA), Oct. 2000.

[37] Mohamed Ali Kaafar, Laurent Mathy, Chadi Barakat, Kave Salamatian, Thierry Turletti, and Walid Dabbous, "Securing Internet Coordinate Embedding Systems," in *Proc. of SIGCOMM*, (Kyoto, Japan), 2007.

[38] Mohamed Ali Kaafar, Laurent Mathy, Thierry Turletti, and Walid Dabbous, "Virtual Networks under Attack: Disrupting Internet Coordinate Systems," in *Proc. of Second CoNext Conference*, (Lisbon, Portugal), 2006.

[39] Brad Karp and H.T. Kung, "Greedy Perimeter Stateless Routing for Wireless Networks," in *Proc. of International Conference on Mobile Computing and Networking*, (Boston, MA), August 2000.

[40] Dina Katabi, Mark Handley, and Charles E. Rohrs, "Congestion control for high bandwidth-delay product networks.," in *Proc. of SIGCOMM*, (Pittsburg, PA), 2002.

[41] Mark Keil and Carl Gutwin, "Classes of Graphs Which Approximate the Complete Euclidean Graph," *Discrete & Computational Geometry*, volume 7, number 1, 1992.

[42] Daniel Kifer, Shai Ben-David, and Johannes Gehrke, "Detecting Change in Data Streams," in *Thirtieth International Conference on Very Large Data Bases*, (Toronto, Canada), August 2004.

[43] Jon Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," in *Proc. of Symposium on the Theory of Computing*, (Portland, OR), May 2000.

[44] Jon Kleinberg, "Bursty and hierarchical structure in streams.," in *Eighth International Conference on Knowledge Discovery and Data Mining*, (Edmonton, Alberta, Canada), July 2002.

[45] Li Lao, Constantine Dovrolis, and M. Y. Sanadidi, "The probe gap model can underestimate the available bandwidth of multihop paths," *SIGCOMM Computer Communication Review*, volume 36, number 5, 2006.

[46] Jonathan Ledlie, Peter Pietzuch, and Margo Seltzer, "Stable and Accurate Network Coordinates," in *Proc. of International Conference on Distributed Computing Systems*, (Lisbon, Portugal), July 2006.

[47] Sung-Ju Lee, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Fonseca Rodrigo, "Measuring bandwidth between planetlab nodes," in *Passive and Active Measurement Workshop*, (Boston, MA), Mar. 2005.

[48] Ben Leong, Barbara Liskov, and Robert Morris, "Geographic Routing without Planarization," in *Proc. of the Symposium on Networked Systems Design and Implementation*, (San Jose, CA), May 2006.

[49] Eng Keong Lua, Timothy Griffin, Marcelo Pias, Han Zheng, and Jon Crowcroft, "On the Accuracy of Embeddings for Internet Coordinate Systems," in *Proc. of Internet Measurement Conference*, (Berkeley, CA), Oct. 2005.

[50] Cristian Lumezanu and Neil Spring, "Playing Vivaldi in Hyperbolic Space," in *Proc. of Internet Measurement Conference*, (Rio de Janeiro, Brazil), Oct. 2006.

[51] Dahlia Malkhi, "Locality-Aware Network Solutions," Technical Report TR 2004-6, School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel, June 2004.

[52] Petar Maymounkov and David Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Proc. of International Workshop on Peer-to-Peer Systems*, (Cambridge, MA), March 2002.

[53] A. W. Moore and J. W. Jorgenson, "Median Filtering for Removal of Low-Frequency Background Drift," *Analytic Chemistry*, volume 65, page 188, 1993.

[54] Akihiro Nakao, Larry Peterson, and Andy Bavier, "A Routing Underlay for Overlay Networks," in *Proc. of SIGCOMM*, (Karlsruhe, Germany), Aug. 2003.

[55] Moni Naor and Udi Wieder, "Know thy Neighbor's Neighbor: Better Routing for Skip-Graphs and Small Worlds," in *Proc. of International Workshop on Peer-to-Peer Systems*, Feb. 2004.

[56] J.A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *Computer Journal*, pages 308–313, September/October 1965.

[57] Eugene Ng and Hui Zhang, "Predicting Internet Network Distance with Coordinates-Based Approaches," in *Proc. of INFOCOM*, (New York, NY), June 2002.

[58] Eugene Ng and Hui Zhang, "A Network Positioning System for the Internet," in *Proc. of USENIX Annual Technical Conference*, (Boston, MA), June 2004.

[59] David Oppenheimer, Brent Chun, David Patterson, Alex Snoeren, and Amin Vahdat, "Service Placement in a Shared Wide-Area Platform," in *Proc. of USENIX Technical Conference*, (Boston, MA), 2006.

[60] Andrew Parker, "File Formats Traversing P2P Networks (CacheLogic White Paper)." `http://www.cachelogic.com/`.

[61] Larry Peterson, Tom Anderson, David Culler, and Tim Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," in *Proc. of Workshop on Hot Topics in Networks*, (Princeton, NJ), October 2002.

[62] Marcelo Pias, Jon Crowcroft, Steve Wilbur, Tim Harris, and Saleem Bhatti, "Lighthouses for Scalable Distributed Location," in *Proc. of International Workshop on Peer-to-Peer Systems*, (Berkeley, CA), February 2003.

[63] Peter Pietzuch, Jonathan Ledlie, Michael Mitzenmacher, and Margo Seltzer, "Network-Aware Overlays with Network Coordinates," in *Proc. of International Workshop on Dynamic Distributed Systems*, (Lisbon, Portugal), July 2006.

[64] Peter Pietzuch, Jonathan Ledlie, and Margo Seltzer, "Supporting Network Coordinates on PlanetLab," in *Proc. of Workshop on Real, Large Distributed Systems*, (San Francisco, CA), Dec. 2005.

[65] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Matt Welsh, Margo Seltzer, and Mema Roussopoulos, "Network-Aware Operator Placement for Stream-Processing Systems," in *Proc. of International Conference on Data Engineering*, (Atlanta, GA), April 2006.

[66] Johan Pouwelse, Pawel Garbacki, Jun Wang, Amo Bakker, Jie Yang, Alexandru Iosup, Dick Epema, Marcel Reinders, Maarten van Steen, and Henk Sips, "Tribler: A social-based peer-to-peer system," *Concurrency and Computation*, May 2007.

[67] Sylvia Ratnasamy, Paul Francis, Mark Handley, Brad Karp, and Scott Shenker, "Topology-Aware Overlay Construction and Server Selection," in *Proc. of INFOCOM*, (New York, NY), June 2002.

[68] "Red Swoosh." `http://www.akamai.com/redswoosh/`.

[69] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz, "Handling Churn in a DHT," in *Proc. of the USENIX Technical Conference*, (Boston, MA), June 2004.

[70] Antony Rowstron and Peter Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *Proc. of Middleware*, (Heidelberg, Germany), Nov. 2001.

[71] Yuval Shavitt and Tomer Tankel, "Big-Bang Simulation for embedding network distances in Euclidean space," in *Proc. of INFOCOM*, (San Francisco, CA), June 2003.

[72] Yuval Shavitt and Tomer Tankel, "On the Curvature of the Internet and its usage for Overlay Construction and Distance Estimation," in *Proc. of INFOCOM*, (Hong Kong), June 2004.

[73] Jeff Shneidman, Peter Pietzuch, Matt Welsh, Margo Seltzer, and Mema Roussopoulos, "A Cost-Space Approach to Distributed Query Optimization in Stream Based Overlays," in *Proc. of International Workshop on Networking Meets Databases*, (Tokyo, Japan), Apr. 2005.

[74] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. of SIGCOMM*, (San Diego, CA), Aug. 2001.

[75] Gabor Szekely and Maria Rizzo, "Testing for Equal Distributions in High Dimension," *InterStat*, volume 5, November 2004.

[76] Liying Tang and Mark Crovella, "Virtual Landmarks for the Internet," in *Proc. of Internet Measurement Conference*, (Miami Beach, FL), Oct. 2003.

[77] Kevin Thompson, Gregory Miller, and Rick Wilder, "Wide-area Internet traffic patterns and characteristics," *IEEE Network*, volume 11, Nov/Dec 1997.

[78] Marcel Waldvogel and Roberto Rinaldi, "Efficient topology-aware overlay network," *SIGCOMM Computer Communication Review*, volume 33, number 1, 2003.

[79] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer, "Meridian: A Lightweight Network Location Service without Virtual Coordinates," in *Proc. of SIGCOMM*, (Philadelphia, PA), Aug. 2005.

[80] Rongmei Zhang, Charlie Hu, Xiaojun Lin, and Sonia Fahmy, "A Hierarchical Approach to Internet Distance Prediction," in *Proc. of International Conference on Distributed Computing Systems*, (Lisbon, Portugal), July 2006.

[81] Han Zheng, Eng Keong Lua, Marcelo Pias, and Timothy G. Griffin, "Internet Routing Policies and Round-Trip-Times," in *Proc. of Passive and Active Measurement Workshop*, (Boston, MA), Mar. 2005.