Internet-scale Visualization and Detection of Performance Events

Jeffrey Pang, Subhabrata Sen, Oliver Spatscheck, Shobha Venkataraman
AT&T Labs - Research, Florham Park, NJ, USA

1 Introduction

Network server farms host a wide range of important applications, such as e-commerce, content distribution, and cloud services. Because server farms serve customers spread across the Internet, the key to effective server farm management is the ability to detect and resolve problems between a farm and its clients. Operators typically monitor performance using rule-based scripts to automatically flag "events of interest" in an array of active and passive measurement feeds. While effective, these rule-based approaches are usually limited to events with known properties. Equally important to operators is finding the "unknown unknowns" — novel events of interest with properties that have not been observed before. Effective visualization greatly aids in the discovery of such events, as operators with domain expertise can quickly notice unexpected performance patterns when represented visually. This paper presents *BirdsEye*, a tool that visualizes performance at Internet scale.

Designing such a tool is non-trivial because operators have to diagnose performance problems that may manifest themselves anywhere on the Internet. Visualizing all the possible ways these problems may manifest themselves poses three challenges: First, the vastness of the Internet and the sheer volume of raw performance data make it impossible for a human operator to comprehend every piece of information about every part of the Internet. An effective visualization needs to be sparse in representation, yet discriminating of good and poor performance. Second, problems can manifest themselves at multiple scales – e.g., a degraded peering link might impact entire swaths of the IP address space while a misdirected client might only affect a single ISP. Thus, there is not a single "level" of monitoring that can capture all problems that operators care about. Finally, performance problems not only correlate across space, but also across time – e.g., a problem may occur periodically during a certain time of the day. Thus, an effective visualization must present both the spatial view of performance and show how it changes over time.

To meet these challenges, we first observe that a tree is a natural way to visualize the Internet performance from the perspective of a server farm. That is, the IP address hierarchy can be interpreted as a tree with each node corresponding to an IP prefix, and its children corresponding to sub-prefixes. If we color a node (e.g., progressively from green to red) based on the likelihood that an IP address in that node's prefix is experiencing a performance problem, we will likely be able to differentiate "good" portions of the address space vs. "bad" portions. This is because IP addresses in the same prefix are more likely to be geographically close, under the same administrative control, and/or share the same routing paths. Thus, their performance is likely to be correlated. The focus of this paper is how to visualize this tree effectively.

A straight-forward approach would be to visualize the entire tree up to a predetermined aggregation-level, such as BGP prefixes. But this approach either would not sparse enough for human comprehension, or would not represent problems at granularities other than the predefined one. Instead, BirdsEye builds *adaptive decision trees* over the IP address space using recent performance measurements. These decision trees group IP addresses with similar performance characteristics and separate those with significantly different performance characteristics. Moreover, these trees are learned online, and adapt to changes in the underlying network. Therefore, changes in performance are reflected in the decision tree over time. By visualizing these adaptive decision trees, BirdsEye shows the performance to the entire Internet, but only highlights the parts that have bad performance at any given point in time.

We present an evaluation of our tool using more then 50 million Round Trip Time (RTT) measurements collected from a distributed server farm in a tier-1 ISP. While RTTs are not the only performance measurements we can visualize, they are one important metric of interest tracked by many operators. Through this case study,

we discover several RTT anomalies, such as diurnal patterns of poor performance in particular access ISPs and an ISP that was likely misdirected by the server farm. This was unknown to operators, suggesting that BirdsEye is indeed useful in finding novel performance problems. We envision that BirdsEye will supplement existing rule based systems — once an operator has verified that a hitherto unknown pattern deserves more attention, they can create new rule-based scripts to flag the patterns.

2 Design Overview

2.1 Design Requirements

The challenges described in Sec. 1 dictate four requirements for visualizing the Internet tree:

Sparse Network-wide Representation. The visualization of the tree needs to encompass the entire Internet in order to able to pinpoint any region with performance problems. However, in order to be usable by human operators, the tree also needs to be sparse, highlighting only the regions needed to differentiate performance experienced by clients. We ensure a sparse representation by enforcing a limit on the maximum number of leaves the Internet tree can have.

Multi-Level Drill-Down. While being sparse, the tree should not overly focus on a single level of the address space such as /24s or /8s since problems may manifest themselves at multiple levels in the hierarchy. For example, our case studies show that there are scenarios where large prefix ranges (e.g., /10 blocks) can be combined because they all experience the same performance. However, there are other scenarios where small ranges (e.g., belonging to management systems) must be identified individually to ensure that their performance is monitored. Since the depth of each branch in the tree represents how far operators can visually drill-down into a given prefix, our tool automatically infers the depth needed to differentiate performance among IPs in each prefix.

Capture Temporal Dynamics. The tree also needs to reflect changes in the performance of clients across the Internet. For example, by looking at a time series of trees, an operator should be able to quickly see prefixes that deviate from normal performance, e.g., due to a degraded peering link. We address this requirement by computing an *adaptive* tree. That is, it can modify its structure and performance indicators over time as more measurements are received.

Real-time Rendering. Finally, the tree needs to be constructed efficiently over large volumes of performance measurements, and updated periodically (e.g. every 5 minutes) to allow operators view the performance measurements in a timely fashion. We address this requirement by ensuring that the tree can be constructed in an online fashion over a stream of performance data.

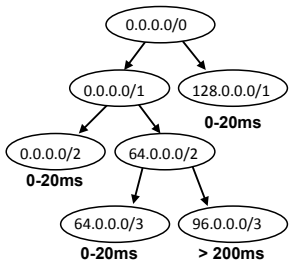


Figure 1: An example IPTree with 4 leaves.

2.2 Tool Overview

BirdsEye has two main components: (1) the *tree-constructor*, which generates Internet trees that meet the aforementioned design requirements, and (2) the *visualizer* which generates visualizations of the tree, in a manner that highlights anomalies and changes in performance. The tree-constructor takes as input a stream of performance data (e.g., RTT measurements). Each time interval, it sends an updated tree to the visualizer. The visualizer then generates and displays a graphic for the updated tree, using each node's performance indicators to colour it. We describe the construction of the tree in detail in Section 3, and visualization in Section 4. In Section 5, we show, using real-world examples, that the time series of Internet trees visually highlights both regular and irregular performance patterns.

3 Generating Internet Trees

In this section, we describe our algorithm for generating accurate Internet trees, focusing on latency measurements as a concrete example of performance data. Since the Internet tree needs to differentiate between client IP addresses based on their performance, it effectively builds a decision tree over the IP address hierarchy. Automatically inferring such an Internet tree from performance data is thus a decision-tree learning problem, quite different from hierarchical heavy-hitter problems (see Sec. 6). Note, however, that this decision tree is quite different from decision trees typically built in diagnosis applications – our tree is based purely on the structure of the IP address hierarchy. This, along with our requirements, make it infeasible to apply standard decision-tree learning algorithms. Instead, we extend the algorithmic framework proposed in [8] for latency prediction at server farms, since this framework incorporates our design requirements (i.e., learning over streaming data, sparse representation, fundamentally adaptive tree, noise tolerance) with theoretical guarantees.

Modeling Design Requirements. We first formally model the design requirements of the Internet tree into its definition (termed *IPTree* to avoid confusion): An *IPTree* T_P over the IP address hierarchy is a tree whose nodes are prefixes $P \in \mathcal{P}$, and whose leaves are each associated with a label for prediction (e.g., a label may be "0-20ms"). An IPtree is thus a decision tree for IP ad-

dresses $\mathcal{I}$: an IP address i gets the label associated with its longest matching prefix in P. We define the *size* of an IPtree to be the number of leaves needed when it is represented as a binary tree. We define an *adaptive k-IPtree* to be an IPtree that can (a) contain at most k leaves, (b) grow nodes over time, and (c) change the labels of its leaf nodes, and (d) reconfigure itself occasionally. Fig. 1 shows an example IPTree with 4 leaves; each leaf is labeled with the latency range associated with its subtree. Our case studies in Sec. 5 show how all of these operations are useful to maintain an accurate Internet tree.

We need to learn an IPtree with high predictive accuracy, as the accuracy reflects how well it models the data. However, standard decision tree algorithms do not meet many of our design requirements; e.g. most learning algorithms assume that data originates from a fixed distribution; however, we cannot make such an assumption as our tree needs to be able adapt its predictions quickly when there are changes in the input data stream. Most decision tree learning algorithms also do not operate on a data stream – they require multiple passes over the data. The properties required of our IPtree learning algorithm from Section 2 are instead naturally captured in the mistake-bound model of learning [5, 6, 8] and so we build on this model for BirdsEye. For visualization purposes, it is sufficient for the tree to predict the latency within an appropriate range, so we split latency into a number of pre-defined categories and require the tree to predict the right category. [1]

Algorithm Sketch. In the learning model of [8], the algorithm is given an IP address to predict on, makes a prediction, and then is given the correct label to update its internal k-IPtree. At a high-level, the algorithm involves all parent prefixes of an IP i in current IPtree in both steps, i.e., making a prediction for i, as well as in updating itself (i.e., learning). The key aspect of the algorithm is to decompose the main prediction problem into 3 subproblems, which can be treated independently: (a) deciding the prediction of each individual parent prefix, (b) combining the parent prefix predictions by deciding their relative importance, and (c) maintaining the tree structure so that the appropriate subtrees are grown and the unwanted subtrees are discarded. The algorithm casts these subproblems as instances of experts' problems [4, 6], a well-explored area in online learning.

We make 2 major changes to extend this binary classification algorithm to operate on a continuous (but categorized) range of latency values. First, each parent prefix now predicts from m categories instead of 2 using the weighted majority algorithm with shifting targets [6] (instead of shifting experts' algorithm in [8]) – this is another experts' algorithm that allows nodes to shift their predictions between categories over time. Second, we penalize incorrect predictions as a function of how far away they are from their respective true latency values; this way, subtrees with latency categories that are farther apart are grown preferentially, all else being equal.

[1] In this paper, we restrict our problem to predicting latency from a set of pre-defined categories. It is possible also to infer the categories automatically by making multiple passes on the data, but we do not consider this extension in this paper.

4 Visualization

Our visualization makes it easy for operators to detect performance changes and determine where in the Internet they occurred. Figure 2 shows an example of an IPtree displayed in BirdsEye. Each dot is a tree node, which represents a specific IP prefix. There are four relevant properties of each node:

- **Node size** is proportional to the log of the number of RTT measurements that it represents. For example, A has 14,000 measurements whereas B has 112.
- **Node color** corresponds to its predicted RTT; green represents low RTT while red represents high RTT. For example, A has a predicted RTT of 0-20ms whereas B has a predicted RTT of 100-200ms.
- **Distance from center** corresponds to prefix length; shorter prefixes are closer to the center of the circle, while longer prefixes are closer to the edge. For example A is a /6 whereas B is a /24.
- **Angular location** is chosen with the IP interpreted as an integer i, i.e, $\frac{360i}{2^{32}}$. For example, A is a prefix of 128.0.0.0 whereas B is a prefix of 142.0.0.0.

The first two properties make it easy to detect changes by giving obvious ques to the size and severity of anomalies. The second two properties make it easy to determine where changes occur by ensuring that the same IP prefix will always appear in the same place in the visualization and that related prefixes are close to each other.

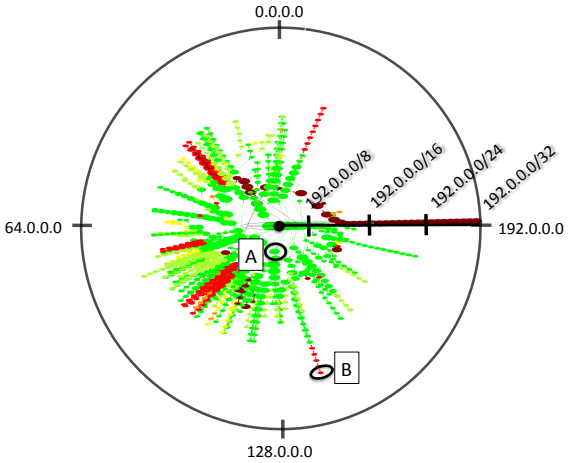


Figure 2: Example of an IPtree displayed in BirdsEye. Each dot is a tree node, which represents a specific IP prefix. Sec. 4 describes how nodes are laid out.

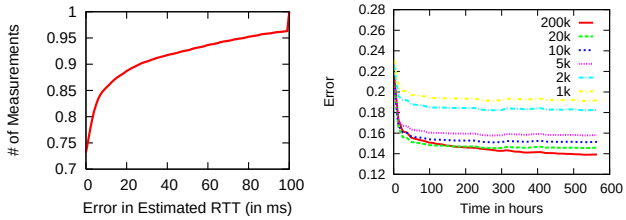


Figure 3: (a) CDF of Error and (b) Misclassified categories as a function of tree size k.

Finally, we must select the parameter k of the IPtree, which determines the number of nodes. Displaying more nodes gives operators more detailed information about network state but may overwhelm them. Displaying fewer nodes presents a more comprehensible picture, but potentially loses important information. Thankfully, the algorithm in Sec. 3 gives us a mechanism to choose the smallest k that doesn't lose much information: We simply choose the smallest k such that the prediction accuracy of the IPtree does not increase significantly with larger k, or drop substantially over time. Sec. 5.1 describes the k selected based on empirical RTT data.

5 Experiments

To demonstrate BirdsEye's utility, we evaluate the accuracy of its IPtree and present simulations and case studies on RTT measurements collected from one node of a large distributed server farm. The node is located near a major metropolitan area in the north-eastern United States. We collected RTT data based on TCP handshake delays using a network monitor on one of the nodes from April 1 to April 20, 2010. 2-3 million measurements are collected each day across all servers at that node.

We implemented BirdsEye with about 3000 lines of C++. Our current unoptimized implementation takes less than 1 minute to generate the IPtree and corresponding visualization for each node in the server farm. Thus, when integrated with an ongoing feed of RTT measurements, BirdsEye can generate near real-time visualizations of network-wide RTT performance.

As discussed in Sec. 3, we split the latencies into categories: $<$20ms, [20ms-40ms), [40ms-60ms), [60ms-80ms), [80ms-100ms), [100ms-200ms), and $\geq$200ms. The categories reflect user perceived performance differences — e.g., an RTT increase from 10ms to 40ms is more noticeable than one from 110ms to 140ms. Nodes change from green to yellow to red as RTT increases.

5.1 Accuracy Evaluation

We now describe our algorithm's accuracy in predicting RTT categories. We note that our goal is not as much to demonstrate a highly-accurate RTT estimation technique, but rather, to show that the tree computed by our algorithm is accurate enough to use for inferring performance via the visualization.

Fig. 3(a) shows the error in estimating the latency category (this is computed as the difference between the actual RTT and the category) for $k = 20,000$, over the entire data set. We note that 83% of RTTs are estimated within 5ms of their category, and 90% are estimated within 20ms, (i.e., to a neighbouring category, at most). Thus, the IPtree's prediction of the latency category is accurate enough for visualizing performance the vast majority of the time.

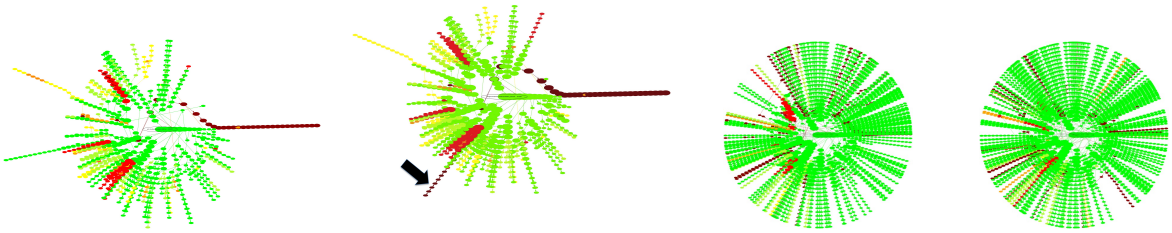
In order to choose an appropriate k for the visualization, we now examine the algorithm's accuracy over time for different k. Fig. 3(b) shows the fraction of misclassified categories over time for k ranging from 1000 to 200,000. Using a tree with $k = 1000$ or $k = 2000$ always produces notably more error than using $k = 5000$ leaves. Increasing k beyond 5000 reduces the error by only by 1% compared to $k = 200,000$. Since visualization is most effective with smaller k, we use $k = 5000$ for the visualization without significant loss in accuracy.

5.2 Injected Anomalies

Next, we use injected anomalies to illustrate Birdseye's ability to visually highlight anomalies. We consider two different anomaly scenarios: one with local performance impact, and another with network-wide impact.

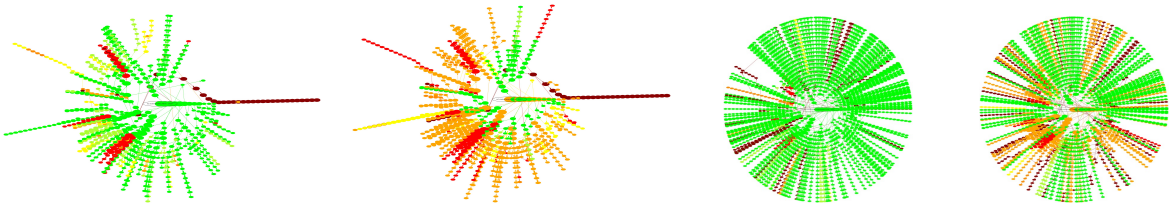
We first consider a class of anomalies that is difficult to detect with traditional methods. Consider a small prefix (e.g., a /24) that is not advertised by itself in BGP (because it is always a part of a larger advertised prefix), and thus would not likely be discovered by examining only advertised BGP prefixes. *How well does BirdsEye handle such a scenario?* We select a /24 prefix of a highly active /13 belonging to a major tier-1 ISP, and add randomly generated IP addresses in this /24 with high RTTs (e.g., 100-200ms) into the stream, such that the injected data is no more than 1% of the parent /13's data. Fig. 4(b) shows BirdsEye IPtree for the hour after this injected anomaly – we see a new red spike (highlighted) corresponding to the anomalous /24, which is not present earlier, i.e., Fig. 4(a). Note that finding such an anomaly even if all /24 prefixes are tracked is not trivial: Figs. 4(c) & (d) show the same anomaly inserted into a strawman tree that consists only of /24 prefixes (but is otherwise grown identically as our tree); the anomaly is lost in Fig. 4(d).

We now illustrate how BirdsEye captures a large-scale performance event. We collect the set of all prefixes advertised by a major tier-1 ISP to the server farm node, and add a delay to each IP address in that set during a time interval. This simulates a scenario where a ISP-wide disruption cause traffic flowing to different destinations through that ISP to be rerouted over much longer paths (e.g., cuts in critical bottleneck links as happened in the Mediterranean Sea in 2008) . Fig. 5(a) & (b) shows



(a) Birdseye: Before anomaly (b) Birdseye: After anomaly (c) Strawman: Before (d) Strawman: After

Figure 4: Injected anomaly: high latency in a /24 prefix block. (a)-(b) show BirdsEye before & after the anomaly, which clearly reveals the /24 block. (c)-(d) show a strawman tree of /24 prefixes, but here the anomaly is lost.



(a) Birdseye: Before (b) Birdseye: After (c) Strawman: Before (d) Strawman: After

Figure 5: Injected ISP-wide performance event: (a)-(b) show BirdsEye before & after the anomaly, which dominates the tree. (c)-(d) show a strawman tree of /24 prefixes, where the anomaly is somewhat less visible.

how this anomaly creates a big visual impact – nearly the entire tree changes in color between the two trees. The strawman tree also changes its color between Fig. 5(c) & (d), but it is less noticeable. Thus, Birdseye is able to visually highlight the impact of a large anomaly as well.

5.3 Real Case Studies

We now present examples of real RTT anomalies discovered using BirdsEye. To illustrate these, we use snapshots of the BirdsEye IPtree at different hours of the day as well as the high-RTT subtrees in Fig. 6.

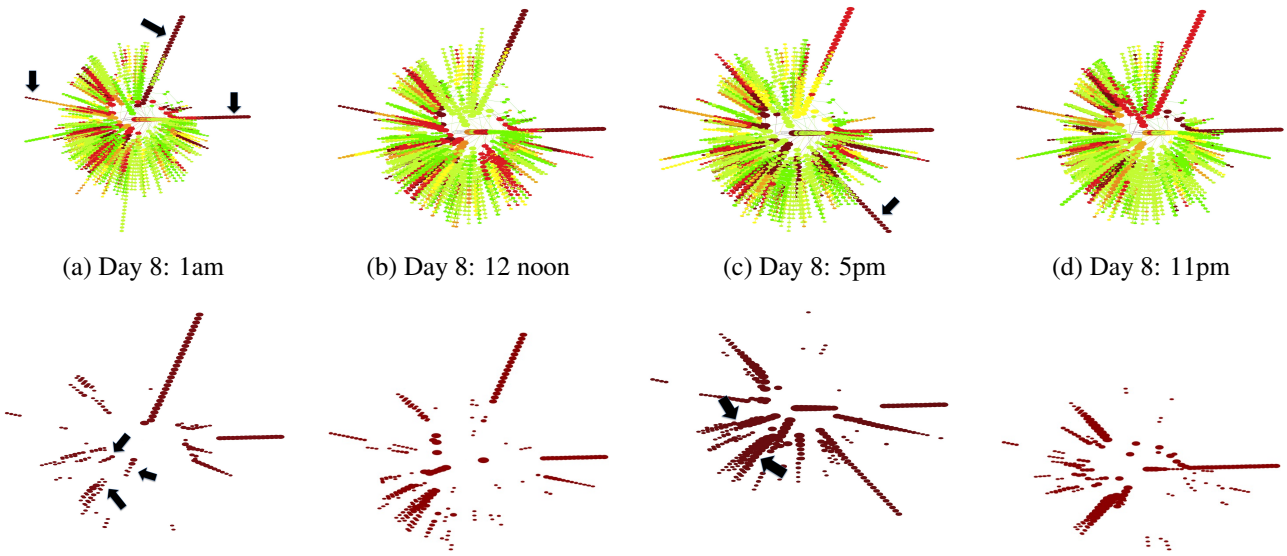
Case Study 1: Consistently Poor Performance. Our first example focuses on parts of the IPtree that always have high RTT (i.e., always appear red). Each snapshot shows 3 consistent long spikes of high RTT in the IPtree (highlighted in Fig 6(a)). On examining these prefixes, we found that they correspond to the management nodes of the distributed server farm located on the West Coast, so their high RTT is not particularly of concern to the server farm's operation. We validated that the IPs do indeed have high RTT by examining the data — $73-81\%$ of the RTTs exceed 90ms, thus justifying their presence.

In addition, there are permanent red areas in all snapshots that are larger prefixes which range from /12 to /18 blocks (highlighted in Fig 6(e)). Recall that larger prefixes are closer to the IPtree's center. Inspection reveals that those prefixes belong to a cellular carrier,[2] and shows that $56-72\%$ of the corresponding RTTs in these prefix blocks are over $80ms$. While high RTTs on wireless carriers are expected, this highlights that those wireless users may have issues accessing latency-sensitive content stored on the server farm.

[2]This carrier is not affiliated with the authors.

Case Study 2: Occasionally Poor Performance. Our second example explores a part of the IPtree that also regularly appears, but experiences high RTTs only occasionally, highlighted in Fig. 6(g). Note that this region shifts from green in the early hours (e.g., Fig. 6(a)) to yellow/red during the busier hours. This is also especially visible from the high-RTT subtrees – these prefixes do not appear in Fig. 6(e), but appear in Fig. 6(g). Closer investigation revealed that most of these prefixes were access ISPs that seem to show signs of congestion during the evening hours, even while there are other ISPs do not (i.e., stay green). Detailed analysis of the measurement data showed that in these access networks, around 40% of the RTTs increased by over $40ms$! Finding the set of all these ISPs using traditional tools, which plot performance per ISP, would have been extremely tedious.

Case Study 3: Anomalous IP block. Our last example shows how BirdsEye may aid in finding an anomaly that may otherwise be lost in noise. We focus on the prefixes highlighted in Fig. 6(c). Note that this spike is not present in the other 3 hours. When we manually examined the prefixes, we discovered that these prefixes belong to a small ISP in the western US, and the IPs appear for 2-3 hours on 3 different days in our data set. They account for less than $0.02-0.05\%$ of the RTTs in those hours, however, 95% of them exceed 80ms, and about 41% exceed 200ms. Even though the IP block comprises a tiny volume of data, BirdsEye differentiates it from a parent prefix with over 100 times more data, 95% of whose RTTs are *under* 80ms. The geographical location of the IP block suggests that these clients were misdirected at the time, as the server farm has nodes that are geographically closer to this ISP. Before BirdsEye, our



(a) Day 8: 1am (b) Day 8: 12 noon (c) Day 8: 5pm (d) Day 8: 11pm

(e) High RTT Subtree: 1am (f) High RTT: 12 noon (g) High RTT: 5pm (h) High RTT: 11pm

Figure 6: Real Case Studies. Figs. (a)-(d) show an IPtree time series through Day 8, and (e)-(h) show the corresponding high-RTT subtrees. Figs. (a) & (e) highlight regions with consistently high RTTs, (g) highlights a region with diurnal pattern and (b)-(d) highlight a one-time misdirected client at the server farm.

operators did not know that this ISP had been directed to this particular node, nor of its extremely high RTT.

6 Related Work

Visualization has been acknowledged as an important way to understand Internet characteristics [2] but an effective visualization requires a compact representation of the data. We focus here on work related to our representation, the Internet tree. Algorithms for building hierarchical heavy-hitter clusters [3, 9] also summarize traffic characteristics into a small number of prefix clusters; however, our problem differs from these as their goal is typically to identify prefixes with substantial traffic, not differentiate performance characteristics. More closely related are approaches that build optimal aggregates [1,7] over the address space to classify traffic with different characteristics; we build on [8] as it is designed for automatically adapting over changing data streams. Our problem also differs from the classic latency estimation problem in networking research: our goal is not to estimate end-to-end latency between arbitrary hosts, but to differentiate latency performance by prefix.

7 Conclusion

We presented BirdsEye, a visualization tool that enables operators to track network-wide performance between a server farm and its customers. It builds *adaptive decision trees* over the IP address space using recent performance measurements, which group IP addresses with similar performance characteristics and separate those with significantly different performance characteristics. By visualizing these decision trees, BirdsEye shows the performance to the entire Internet, but only highlights the parts that have bad performance at any given point in time. As a case study, we used BirdsEye to visualize RTT measurements for a commercial server farm, and discovered several RTT patterns that operators were unaware of but were keenly interested to know, such as diurnal patterns of poor performance in particular access ISPs and an ISP likely misdirected by the server farm. Our approach is likely to be useful in any application where differences in behaviour depend upon the IP address structure.

References

[1] BEVERLY, R., AND SOLLINS, K. An internet protocol address clustering algorithm. In *SysML* (2008).

[2] BURCH, H., AND CHESWICK, B. Internet watch: Mapping the Internet. *Computer 32*, 4 (Apr. 1999), 97–98.

[3] ESTAN, C., SAVAGE, S., AND VARGHESE, G. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM* (2003).

[4] FREUND, Y., SCHAPIRE, R. E., SINGER, Y., AND WARMUTH, M. K. Using and combining predictors that specialize. In *STOC* (1997).

[5] LITTLESTONE, N. Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning 2*, 285-318 (1988).

[6] LITTLESTONE, N., AND WARMUTH, M. The weighted majority algorithm. *Information and Computation 108* (1994), 212–251.

[7] SOLDO, F., MARKOPOULO, A., AND ARGYRAKI, K. Optimal filtering of source address prefixes: Models and algorithms. In *INFOCOM* (2009).

[8] VENKATARAMAN, S., BLUM, A., SONG, D., SEN, S., AND SPATSCHECK, O. Tracking dynamic sources of malicious activity at internet-scale. In *NIPS* (2009).

[9] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *IMC* (2004).