Trajectory Engine: A Backend for Trajectory Sampling

N. G. Duffield A. Gerber M. Grossglauser

AT&T Labs - Research
180 Park Ave, Florham Park NJ 07932, USA
{duffield, gerber, mgross}@research.att.com

Abstract

The management of communications networks increasingly requires detailed knowledge of network usage, acquired by direct measurement. We report on the design and implementation of a backend system for Trajectory Sampling, a method for the consistent sampling of packets in transmission across a network domain. This Trajectory Engine collects trajectory samples and stores them after appropriate preprocessing. It provides a querying and visualization tool to aid in traffic engineering and troubleshooting tasks.

We describe the entire system in detail, and in particular the design choices that we took in order to balance the scale of system (due to large volumes of measured data) with resource usage while providing useful functionality for users. In the preprocessing stage, we focus on reassembly of trajectories from individual samples. We describe the design of the database for efficient storage of trajectories, and its relationship with the query and visualization interface. We test the system using a synthetic stream of trajectory samples derived from configuration and usage data from the network of a major service provider. We walk through several examples that illustrate how a network operator might take advantage of trajectory sampling through such a tool.

1 Introduction

1.1 Background

The efficiency of resource allocation and the quality of service provided by IP networks depend critically on effective traffic management. Control and engineering functions rely on a characterization of the traffic that flows through the network [5, 6]. Traffic measurement is an integral component of these functions, both to provide input to these functions, and to verify that their actions are having the desired effect. Virtually all traffic engineering functions, such as route optimization or planning of failover strategies, rely on an understanding of the spatial flow of traffic through the domain. For example, suppose we observe that some link in the backbone is overloaded. Appropriate corrective action requires an understanding of which ingress points the traffic observed on this link originates and where it is headed, what customers are affected by the congestion, and what the traffic mix is; without this information, effective remedies (e.g., rerouting of part of that traffic) cannot be taken. However, traffic flow measurements available today are typically highly aggregated (e.g., router reports of link utilization), or indirect, relying on knowledge of the current network state (e.g., routing and/or forwarding tables) for their interpretation. Moreover, increasing link speeds make it infeasible to measure and report on all flows individually.

We have recently proposed a new method for direct traffic observation called trajectory sampling [4]. A subset of packets are selected on every link through which they pass. These packets can then be used as a representative of the overall traffic. If packets were simply randomly sampled at each link, we would be

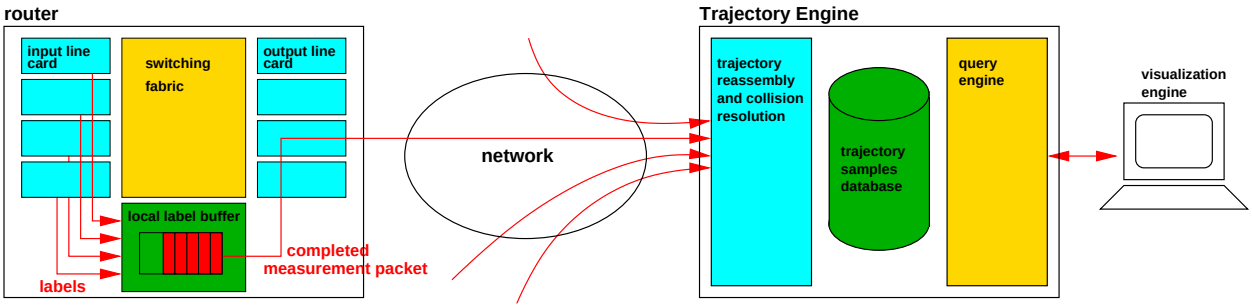

Figure 1: The overall system consists of a measurement subsystem on every router in the network domain, and the Trajectory Engine, a central measurement collection and processing system.

unable to derive the precise path that a sampled packet has followed between the ingress to the egress point. Instead, the sampling decision is based on a deterministic hash function over the fields of the packet that remain invariant as the packet traverses the network. This ensures that a packet is either selected on every link it traverses, or on no link at all; we call this trajectory sampling. The proportion of packets selected can be determined by tuning the parameters of the hash function.

A router sends a report, a trajectory sample, to a collector on each packet that it selects. Packets comprise header information—such as the source and destination IP address (used to route the packet) or TCP or UDP port numbers (used by end hosts to direct the packet to the correct application)—followed by the payload. It is the header information that we wish to report to the collector. We can compress the volume of such reports by using another hash over the packet to produce a label. When the packet is first sampled (at the ingress point), both its label, and header fields of interest are reported. Given this association, only the labels need be reported thereafter, i.e. from each interior link that the packet traverses.

As examples of the utility of the approach, the analysis of trajectory samples gathered from all routers in a network domain could determine the set of paths taken by packets between a given source and destination during a period of routing instability, or be used to continuously monitor and identify the links at which packet loss occurs on any given network path. These detailed analyses are infeasible to perform using measurements currently available from production networks.

1.2 Contribution and Requirements

The realization of trajectory sampling requires a measurement subsystem on every router, and a central measurement collection system; see Fig. 1. In this paper we report on the design and development of a prototype system for the collection and analysis of trajectory samples. We call this system the Trajectory Engine. Our previous work [4] focused on the mechanisms by which samples would be generated at network elements such as routers. The motivation of the present work is to demonstrate, by construction of the Trajectory Engine, that a stream of trajectory samples, as produced by the routers in a representative network domain, can be used to perform complex analyses such as those mentioned in the previous paragraph.

To this end, the Trajectory Engine performs the following functions. First, it collects the trajectory samples that have been generated by routers (or interfaces cards) and transmitted to it across the network. Second, it reconstructs complete trajectories from these samples, and stores them in a database. Third, it offers querying primitives on the database for use by management and control applications. Fourth, it furnishes a graphical user interface (GUI) for the interactive formulation of queries and presentation of results in a visually useful form. In order to fulfill these requirements, we had to identify and solve a number of design problems. We explain these in detail, and describe our solutions in Section 2.

One challenge is the presence of label collisions: trajectory samples from different packet can have the same label. The presence of collisions can, in some cases, prevent reconstruction of unique trajectories. Thus we need strategies to control the rate of collisions, identify their occurrence, and limit their effect on the accuracy of database queries. Section 2.1 analyzes the frequency of collisions, and describes two approaches by which they can be handled, and their statistical effects compensated for. Section 2.2 describes a timer-based method by which samples can be grouped into complete trajectories.

The second challenge is scalability. The collection system has to perform preprocessing functions (trajectory reconstruction and collision resolution) in real time; the database of trajectory samples must accommodate potentially huge volumes of accumulated data, and certain queries may require complex data aggregations. Section 2.3 describes the trajectory sample database, and the structural decisions taken to facilitate both fast querying, and the interaction with network configuration data used to interpret the trajectories.

The third challenge is the flexibility and generality of queries that can be performed on the database of trajectory samples. Trajectory samples are very general, allowing for a wide range of queries However, storing raw trajectory samples may be prohibitive because the volume is too large. In this case, it will be necessary to sacrifice some of the generality of queries to save space, by preaggregating samples before they are stored in the database. Clearly, there is a tradeoff between scalability and generality of queries. Section 2.3.5 describes the set of querying primitives, and their employment through the GUI.

The testing and evaluation of the Trajectory Engine raises another challenge. Since trajectory sampling is not currently implemented in production routers, we need to furnish a synthetic stream of samples representative of those that would be generated in an actual network. In Section 3 we describe how this is done using supplementary network configuration and measured usage data from the backbone of a major network service provider. We demonstrate three examples of queries supported by the Trajectory Engine.

2 Design

2.1 Label Collisions and their Handling

2.1.1 The Frequency of Collisions

As discussed in our earlier work [4], several packets may happen to have identical labels. Here we summarize the analysis of the proportion of labels affected by collisions. Assume there are trajectory samples from n distinct packets, each sample carrying a label of m bits. Then the alphabet size of the identification hash is 2^m. We wish to determine the number of unique labels , i.e., those that are present in samples from exactly one packet. Assuming label values to be taken with equal probability, it can be shown that this number is, on average, $U(n,m) = n(1-2^{-m})^{n-1}$. Consider now the problem of maximizing the number of unique labels under a given constraint on their total volume $c = nm$. Thus we seek to maximize $U(n) = U(n, c/n)$ over n. Let n^* denote the maximizer and set $m^* = c/n^*$. In [4] we establish the asymptotic behavior:

$$m^* \sim \log_2 c \quad \text{and} \quad n^* \sim \frac{c}{\log_2 c}\,, \quad \text{as } c \to \infty \tag{1}$$

The probability that a given label is in collision in this case is

$$p_{\text{coll}} = 1 - \frac{U(n^*)}{n^*} \sim 1 - e^{-1/\log_2 c}\,, \quad \text{as } c \to \infty \tag{2}$$

2.1.2 Compensating for Collisions

As defined in [4], a label subgraph associated with a label l and a measurement interval records the number of times label l is observed on every link in the network within that interval. In the case of a collision, i.e.,

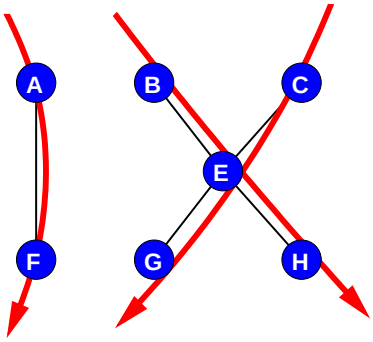

Figure 2: Without compensation, we would overestimate the relative traffic rate on AF, and underestimate BH and CG.

when the label subgraph consists of a superposition of at least two trajectories, it is sometimes possible to disambiguate these trajectories. This is true when the set of component trajectories do not intersect each other. If a label subgraph contains intersecting trajectories, it may not be possible to disambiguate them [4].

A naive approach to handle collisions would be to simply keep trajectory samples that can be disambiguated, and to discard the rest. However, this approach is problematic, because it can introduce bias into estimators. Consider the scenario depicted in Fig. 2. There are three traffic flows in this topology with identical intensities. We try to estimate these flows and their intensities from trajectory samples. Now note that a sample of the flow AF can always be disambiguated, regardless of what label subgraph it is a part of. This is because AF does not intersect other trajectories. Therefore, we could estimate the rate of AF without bias simply by multiplying the observed number of samples of AF with a normalization constant (the inverse of the thinning ratio.) However, for every subgraph where there is at least one sample BH and one sample CG, we would discard the corresponding samples because of ambiguity. Therefore, it is clear that if we estimate the intensities of these two flows in the same way as for AF, a negative bias would result.

To avoid this bias, we would have to renormalize our measurements appropriately. Specifically, we would have to correct for the probability that a trajectory is part of an ambiguous configuration. This is difficult, because this probability depends on the (unknown) rates of all the trajectories in all the ambiguous configurations. The problem can be formulated as a linear inversion problem. We omit the details of this approach; an analogous problem arising in network tomography is described, for example, in [10]. While conceptually feasible, this approach is computationally very demanding, due to the high dimensionality of the problem. Instead, we have pursued another approach to avoid bias, described in the next section.

2.1.3 Exclusion of Labels in Collision

The previous section outlined how to compensate for label collisions by estimating the underlying frequencies of trajectories. An alternative approach is to dimension the measurement system so as to ensure that the frequency of collisions is small. All labels in collision are then identified and excluded from further consideration, whether or not such collisions would give rise to ambiguities. This approach is blunter than that of the previous section in that we make no attempt to extract any information from colliding labels. However, since colliding labels are excluded independently of whether they lead to ambiguities, exclusion is not biased by topology.

Collisions can be identified from ingress trajectory samples alone. If more than one ingress trajectory sample carries the same label in a given time window, then all trajectory samples with this label are excluded from the window. This bias arising from this exclusion can be compensated for through statistical modeling.

The simplest model is to assume that labels are uniformly distributed and that collisions occur independent of the label. For this model we compensate by adjusting all trajectory frequencies upwards by a factor $1/(1-\widehat{p}_{\text{coll}})$, where $\widehat{p}_{\text{coll}}$ is the measured collision frequency.

In order to be confident that this approach is feasible, we need to show that the collision probability is quite small for a reasonable set of network parameters. Consider a collection of trajectory samples from a domain across which packets have a typical hop count h. Let the samples be directed to a single collector across a link of bandwidth w, and consider label collisions occurring during a window of duration τ. Then the capacity $c = nm$ available for n samples of size m bits is roughly $c = w\tau/h$. We assume $w = 10^9$ bits/s (roughly equivalent to an OC48 link), with $\tau = 10s$ (a upper bound for the typical domain transit time), and $h = 10$. According to (2), this yields a collision probability p_{coll} of about 3% when n and m are chosen as n^*, and m^*, those values for which the expected number of unique labels is maximized.

Due to the form (2), p_{coll} is quite insensitive to the assumptions; changing w, τ or h by a factor of 100 changes it by only about half a percent. Increasing τ beyond the domain transit time would increase c and hence allow the reduction of p_{coll}. However, this also increases the cost of determining duplicate labels, due to the increase in the number of ingress samples that must be checked.

We find that collision probabilities can be lowered significantly if the expected number of unique labels is allowed to be submaximal. In the example, $m^* = 30$; increasing m by 10% decreases p_{coll} by roughly an order of magnitude. The number of unique samples is likewise decreased by about 10%; correspondingly, the variance of statistical estimators–based on counts of unique labels–will increase by about 10%. Since the collision probability is so sensitive to the choice of label size m, we recommend that it be chosen conservatively, e.g., as above by first finding the optimal value m^* value based on the largest expected traffic volumes, then taking the actual value m to be somewhat larger.

We now determine the corresponding sampling probability for the example. Assume the samples are drawn from a network comprising ℓ links of capacity w. The total number of unique packets present in the network during a window of time τ is no more than $N = \ell w\tau/(bh)$, where b is the typical number of bits per packet. Taking $\ell = b = 10^4$ we find $N = 10^9$. Thus we would require a sampling rate no greater that about n^*/N, i.e. about 1 in 30, for a fully utilized network. In practice, we may want to employ a smaller sampling rate in order to control consumption of processing and storage resources at the trajectory engine.

2.2 Trajectory Reconstruction

In the previous section, we have assumed that the label subgraph associated with a label and a time slot is known completely. In the absence of any delays experienced by packets and measurement reports in the network, the label subgraphs would be trivial to assemble: we would just discretize time at the desired granularity, and count the number of times a label value l is reported for each link in a given time slot.

In practice, packets experience transmission, propagation, and queueing delays as they travel through the network. Also, samples are subject to various delays: labels are delayed in the router's label buffer if measurement packet can transport multiple labels; once it is sent out, the measurement packet then experiences delay during the transport from the router to the collection system.

In this section, we focus on how to reconstruct trajectories when labels arrive at the collection system subject to random delays (cf. Fig. 3). We introduce the following notation. The measurement domain is a directed graph $G(V, E)$. A trajectory is a path, i.e., an ordered set of links $(e_1, \ldots, e_n)$ in the measurement domain G. $\mathcal{T}$ denotes the set of all valid trajectories. We make the following assumptions. Consider a link e connecting two routers $s(e)$ and $d(e)$. First, we assume that for every link e in the network domain, we know an upper bound $T(e)$ of the total delay a packet can experience. Specifically, $t_{d(e)} - t_{s(e)} \leq T(e)$, where $t_{s(e)}$ and $t_{d(e)}$ are the time instants when some packet arrives at $s(e)$ and at $d(e)$, respectively. Note that in general, $T(e)$ depends on the link capacity and length, the buffer size feeding the link, etc. Second,

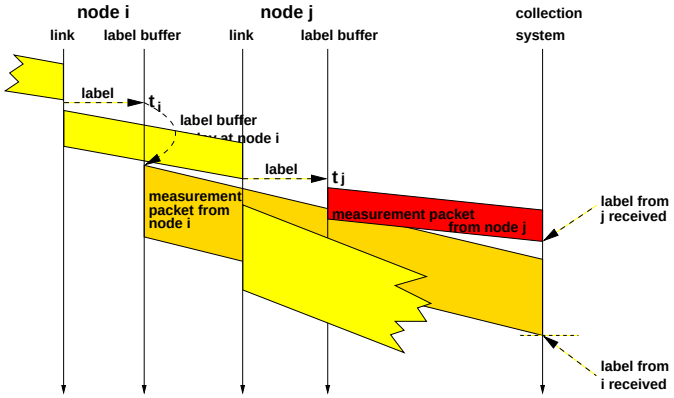

Figure 3: Components of the delay experienced by samples pertaining to a single packet.

we assume that the labels of sampled packets are available immediately, and we assume that the time the label is delayed within the label buffer is bounded by T_b. This can be enforced through a local timer that ensures that incomplete measurement packets get dispatched after at most T_b (such a situation can arise when traffic through a router is very light.) Third, we assume that T_m is an upper bound of the time it takes a measurement packet generated by a router to reach the collection system.

Under these worst-case assumptions, we can devise simple timer-based schemes at the collection system to reassemble trajectories. The basic requirements are (a) to faithfully reconstruct trajectories despite the random arrival times of its labels, and (b) to minimize collisions, i.e., collapsing more trajectories than necessary into label subgraphs (which we discard, as discussed in Section 2.1.3.)

The simplest reconstruction scheme called *trajectory-based timeout* would associate a timer with every possible label value l (in practice, the timer would only have to be instantiated once a label of value l actually arrives.) The timer is started whenever a label l is received at the collection system. All arriving labels l are then accumulated into a trajectory, until the timer times out.

Let us examine what the timeout value should be to ensure that a trajectory does not get truncated. Consider a trajectory $p = (e_1, \ldots, e_n) \in \mathcal{T}$. Assume w.l.g. that a packet is received at the ingress node $s(e_1)$ at $t_{s(e_1)} = 0$. For every link $e \in p$, the time $\tau_{s(e)}$ when the label is received at the collection system satisfies $\tau_{s(e)} \in [0, T_b + T_m]$; the packet arrival time at the next router $d(e)$ satisfies $t_{d(e)} \in [t_{s(e)}, t_{s(e)} + T(e)]$. Therefore, $\tau_{d(e)} \in [0, T_b + T_m + \sum_{e' \in \{e_1,\ldots,e\}} T(e')]$.

We must ensure that for all possible trajectories, the timeout is longer than the difference between the receive times τ of any two labels pertaining to the same packet. This is achieved by setting the timeout to

$$T_t > T_b + T_m + \max_{p \in \mathcal{T}} \sum_{e \in p} T(e) \tag{3}$$

However, this approach suffers from the following problem. Consider two packets A and B that map to the same label value l. Assume that the timer associated with l is started by the arrival of a label of A, and that B arrives at its ingress node before that timer times out. Furthermore, assume that some of the measurement packets that report B from interior nodes arrive before the timeout, but its ingress measurement packet arrives after the timeout. In this case, we would not be able to detect the packet collision (which relies on duplicate ingress reports, as discussed in the previous section), and the partial trajectory would be wrongly attributed to the first packet A only. While in most cases, it would be straightforward to discard this partial trajectory by other means (e.g., considering the network topology), this is undesirably complicated.

We therefore propose a more fine-grained, *link-based timeout* approach. The timer associated with label l is (re)started *every time* a label l arrives at the collection system. As before, when the timer times out,

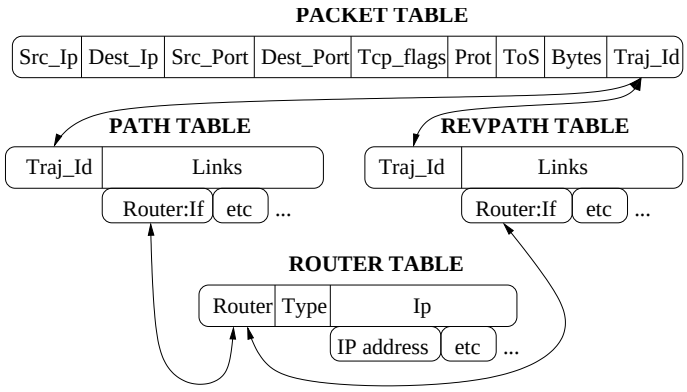

Figure 4: The relational representation of trajectory samples requires four tables.

the trajectory is considered complete. This approach is slightly more complex in that every arriving label requires a timer reset; however, this approach avoids the "edge effect" of the trajectory-based approach, i.e., a packet cannot partially leak into a label subgraph.

The timeout T_l must be chosen to ensure that for any network link e, the time difference between the arrivals of labels from the endpoints $s(e)$ and $d(e)$ of e is less than the timeout. This must hold for all links:

$$T_l > T_b + T_m + \max_{e \in E} T(e) \tag{4}$$

The link-based timeout approach ensures that every sampled trajectory is completely included into exactly one label subgraph. However, there is no upper bound on the number of trajectories that could potentially be included into a label subgraph. In normal operation, the distribution of the number of trajectories included into a single label subgraph falls off geometrically. Only in a situation where a particular label value occurs unusually often (as, for example, in an attack on the measurement system with packets specifically designed to have identical labels) could a label subgraph fail to be complete because the link timer never times out. We guard against this with a cutoff timeout associated with the label subgraph, which times out after a large multiple of T_l.

2.3 Storage, Querying and Visualization of Trajectories

In this section we discuss more fully the requirements for systems to implement the trajectory database. In practice we were able to meet these requirements through use of in-house data management systems that had been created with applications such as Trajectory Engine in mind: the Daytona data management system [7], and the BRAVO data visualization tool [5]. We review their capabilities in this Section.

2.3.1 Tradeoffs and Requirements

Certain trade-offs must be weighed in deciding a format for the trajectory database. These must be evaluated in the context of potentially large data volumes: in the example of Section 2.1.3, then even with a mean sampling probability of 10^{-4}, the Trajectory Engine should be prepared to accommodate up to 10^4 reconstructed trajectories per second when the network is fully loaded.

The first tradeoff is of resources for preprocessing vs. query speed. Storage of reconstructed trajectories in a flat file will be fast; on the other hand user queries will be slow to execute. In view of the data volumes that we anticipate, we require the option of employing indexing to speed up queries.

The second tradeoff is generality of queries vs. data volume. Preaggregation of trajectories could speed up specific queries, but would reduce the set of possible queries. Our main motivation in building the trajectory engine was to field complex queries on traffic patterns. For this reason, we favor generality. In fact, the only data reduction that we employ is to discard timestamps and instantiate new data files every minute. There is virtually no loss of querying generality in practice, since the timeout employed in trajectory reconstruction is of this order.

2.3.2 The Daytona Data Management System

Fortunately, there was a database system available that met our requirements: the Daytona data management system [7]. Daytona offers several definitive advantages recommending its use in the Trajectory Engine. The primary advantage is speed. Daytona translates its Cymbal high-level query language completely into C, which is then compiled into a native machine executable; most other systems interpret their SQL. The Cymbal query language is a very powerful and expressive 4GL that synthesizes a procedural language with several declarative languages including SQL. Its ability to express complex queries, together with its parallelism, enable its effective use of a suitably endowed storage server. In contrast to the server processes acting as operating systems used by other data management systems, Daytona uses Unix as its server, thus saving on code and complexity. Daytona reduces disk usage as compared with other data management systems through a combination of partitioning and compression techniques. This is a great advantage for an application such as the Trajectory Engine, since the amount of data to be stored is potentially enormous. Daytona's horizontal partitioning features allows us to aggregate trajectories over time-slices, in this case of one minute duration. Daytona is currently in applications managing tables containing 75 billion records, added at an average rate of at least 3,000 per second.

2.3.3 Data Storage Tables

Packet trajectories are stored using four tables; see Figure 4. Each packet for which a trajectory is identified gives rise to an entry in the PACKET table, comprising the header information as reported by the ingress router at which the packet was incident. These are the source and destination IP addresses and port numbers, TCP flags (if any), IP protocol, Type of Service, and packet size in bytes. Since many packets may follow the same trajectory, we do not enumerate the packet's trajectory in the packet table. Instead we include there only a trajectory ID, which is a key into the PATH table, whose entries specify observed packet trajectories. Each such trajectory is specified by one of more links, a link comprising two objects: an identifier for a router, and an identifier for an interface on the router. For certain types of query we find it useful to create the REVPATH table, comprising the set of reverse paths of the entries in PATH. The router identifier is a key to the ROUTER table, each entry of which specifies a router type (used to indicate whether the router is at the network core or edge), the router's loopback IP address, and lists the IP addresses of the router's interfaces. The ROUTER table is populated using independent network configuration data.

2.3.4 Data Visualization and Querying

We required a GUI to the trajectory samples which would provide an interactive network map that could be used to specify database queries and present their results. The construction of such systems required a large development effort. Rather than starting from scratch, we were able to reuse and adapt an existing tool that has been developed in house for related purposes. BRAVO [5] was developed to assist network operators in visualizing network topology and network traffic flows. BRAVO's underlying data model comprises objects, such as routers and links, and demands, which are traffic intensities associated with subsets of objects, e.g., between given ingress and egress routers. Using the GUI, users may select and/or aggregate demands (e.g.

all demands sourced by a given customer) and display their intensities by modifying visual attributes in the GUI of the associated objects (e.g. the width or color of routers or links that carry the demands). What-if capabilities allow the user to determine the impact for network load of alternate patterns of demand due e.g. to routing changes or link failures.

BRAVO's underlying data model is very flexible. In particular one can model new types of demands. The facility to select, query and display demands is a class property, with which a demand type is hence automatically endowed. Consequently, we were relatively quickly able to tailor BRAVO to our needs. For our application, a demand is a traffic intensity associated with a trajectory. The demands are calculated in response to user queries at the GUI. The queries are translated to Daytona queries, whose execution returns the appropriate demands to BRAVO. We now describe the data query primitives employed.

2.3.5 Database Query Primitives

Queries to the trajectory engine are specified by three types of arguments: those relating to the paths, packet headers, and times. A query returns a list of trajectories for which packets were found obeying the restrictions specified in the query arguments, together with the aggregate number of packets and bytes of packets that followed each trajectory.

(i) *Path Arguments:* specify a set of links that result in the selection of a set of trajectory identifiers from the PATH table. They comprise a qualifier that takes one of three values, followed, possibly, by a list of links. The qualifier *oneSelected* causes selection of those trajectories that contain one link from the list. The qualifier *allSelected* causes selection of those trajectories that contain all links in the list. The qualifier *all* is null: it imposes no restriction on the set of trajectories.

(ii) *Header Arguments:* restrict attention to packets whose header fields occupy certain ranges. Individual values for all header fields may be specified. In addition, 32 bit masks for the IP source or destination addresses may be specified, allowing selection based on IP address prefix.

(iii) *Time Arguments:* specify the start and end of the interval of times over which packets may be selected. Recall, this time is based on arrival of the ingress trajectory sample at the trajectory engine, expressed at a one-minute granularity.

3 Simulation Results and Application Examples

We describe experiments designed to test the Trajectory Engine under realistic conditions and to illustrate its use in common traffic engineering tasks [2, 5]. Since trajectory sampling is not currently implemented in production routers, we did not have a ready-made stream of trajectory samples at our disposal. Instead, we created a data-driven simulation that generates a synthetic stream of trajectory samples.

The data driving the simulation comprises configuration data (topology, routing) and flow measurements from the backbone of a major service provider. This data is used to generate a stream of events corresponding to the incidence of packets at every router in the network. Thus, both the network topology and the traffic at a coarse granularity of interest for traffic engineering (minutes to hours) is realistic. Only the generation of fine-grained labels relies on assumptions embodied in the simulation. We briefly state these assumptions.

A measured traffic *flow* at a router is a record describing a set of packets with some common property (here, identical source and destination IP address) that traverse an interface of that router. Furthermore, this set of packets must appear close together in time [3]. The record contains a range of fields describing the common characteristics of the flow (source and destination address, port numbers, etc.) and aggregate metrics over the flow (total number of packets and bytes, start and end time, etc.)

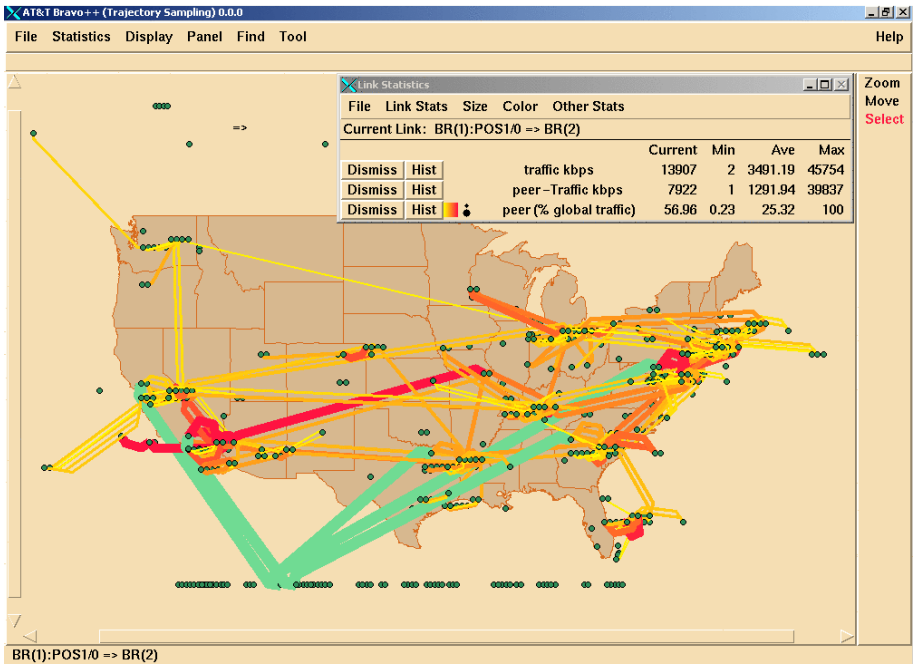

Figure 5: A graphical representation of the flow of traffic from another ISP. The link color represents the percentage of traffic from that ISP on the link.

For our simulation, we have flow measurements available from a subset of ingress links into the backbone network. In order to generate labels from these measurements, we proceed as follows. First, we need to map the measured flows onto the network topology. We do this by joining the flow data with topology and configuration data (such as link weights available to the OSPF routing protocol) to determine the path followed by every flow. Second, we assume that the packets making up the flow are uniformly spaced between the flow's start and end times[1]. Third, we select each packet of the flow independently with probability p, where p corresponds to the desired sampling rate. Fourth, for the packets selected for sampling, we generate a random label. This label appears at every router along the path of the packet. We make the simplifying assumption that packets are not delayed; in other words, the label corresponding to a packet concurrently appears on all the routers on the path. It would be straightforward to refine this model to include random packet delays between routers.

We perform a sanity check of the simulated trajectory sampling information as follows. We have available the link utilization, averaged over 5-minute periods, for every backbone link. This information is obtained through polling of standard SNMP/MIB-II counters [9]. We compare this measured link utilization to that predicted by flow measurements, by aggregating all the flows traversing a given link. We find good correspondence between the two measurements, which suggests that the simulated traffic is a good approximation of the real traffic.

The label stream is fed into the Trajectory Engine described in the previous section. We note that, of the data driving the simulation, the topology data alone is used by the Trajectory Engine; it has no knowledge of the routing state of the network. All the traffic metrics described below are derived from trajectory samples alone. We next describe three examples of how the Trajectory Engine assists a network operator in common traffic engineering tasks.

[1]It would be straightforward to simulate other arrival processes, e.g., Poisson.

Example 1: Studying the impact of peering traffic. The flow of traffic between two ISPs that have a peering relationship depends in a complex way on interdomain routing policies implemented by these and other ISPs [8]. For effective traffic engineering, it is therefore important to be able to examine peering traffic in detail, and to infer how that traffic flows through the network domain.

Figure 5 illustrates how this can be achieved using the Trajectory Engine. First, the set of links that connect to the peer ISP of interest are selected; this can be done either through the GUI, or by a search on the table of links (this is a standard BRAVO capability.) In the query window, we set the path argument to *oneSelected* (cf. Section 2.3.5), which will restrict the trajectories to those traversing (at least) one of these peering links. After also selecting the time interval of interest, launching the query results in a representation as shown in Fig. 5. This representation is useful for a network operator to verify if peering traffic is routed as expected through the domain, to diagnose to what extent congestion problems may arise as a result of the load generated by this particular peer, and assess where the peering traffic encounters performance problems.

An example of a concrete and difficult problem arising in peering is route flapping. Route flapping due to fluctuation in routing tables in peer networks is suggested when traffic originating at a given external IP source address is incident through multiple peering points. At long time-scales this may be indistinguishable from load balancing. However, using the Trajectory Engine to compare the distribution of such traffic inside the network in successive intervals of the time-scale of minutes can reveal such flapping. This approach is even more powerful for detecting flapping of routes entirely in the domain, since these will not be detectable from measurements made at the network boundary.

Example 2: Focusing on a particular application. In this example, we do not restrict the link set with which trajectories have to overlap. Rather, we focus on a particular application. Specifically, we focus on http requests from web browsers (or web proxies) to web servers (or other web proxies). We achieve this by selecting destination port 80 in the query window, because this is the default destination port for the http protocol. This kind of query would be useful to detect and diagnose problems with web hosting centers, web proxies, or with content distribution networks (CDNs), which dynamically redirect queries based on the current load distribution and on the location of the client.

Example 3: Analyzing an overloaded link. Our last example shows how the Trajectory Engine can be used to "drill down" into a problem. Packet loss at a given router is determined by subtracting from the total traffic on inbound links to the router, the traffic on outbound links, after removing from consideration traffic whose source or destination address includes either the router's loopback address, or one of the router interface addresses. Clearly, the proportion of lost packets can be determined from an algebraic combination of the appropriate demand intensities. The tool can also be used to display packet loss at all routers in a single picture, with loss rate indicated by node color. Such a picture gives an immediate high-level view of network performance. Once problematic links and/or routers have been identified, one can focus on traffic traversing the problematic links and routers, and find out its origin and destination, and decide on appropriate corrective action, such as adjusting routing state in order to reroute to a less congested path.

4 Conclusions and Further Work

This paper reported on the Trajectory Engine, and prototype back-end system to receive trajectory samples from network elements, reconstruct and store packet trajectories in a database, and furnish queries through a GUI. The goal was to demonstrate how such a system could provide network operators with new diagnostic tools not available with current network measurements. The main technical challenges arose from the effects of label collisions in samples, the potentially enormous amounts of raw data involved, and the need to find

a satisfactory compromise between query generality on the one hand, and data volumes and query speed on the other. The rationale for the design choices that we employed to meet these challenges constitute the main work reported here. These were: how sampling parameters could be chosen so as to control both sample volumes, and the frequency of label collisions; a timer-based mechanism for the reconstruction of trajectories; and the systems and configuration employed for data management and querying.

Work in progress is to develop a more sophisticated simulation environment, centered around `ssfnet`[1]. This will enable the generation of a more realistic stream of synthetic trajectory samples under which to test trajectory reconstruction. This stream will more accurately reflect transmission latencies of trajectory samples, loss and latency of sampled packets in the network, and can use packet sampling and labeling based on packet content, rather than a probabilistic model.

Further work in progress is developing a method to compensate for label collisions after disambiguation, as outlined in Section 2.1.2. This holds the potential to compensate for more frequent label collisions, thus extending the rate at which samples can be usefully collected.

So far we have assumed that trajectory samples are transmitted reliably to the engine, for example using a reliable transport protocol such as TCP. This has ramifications for the design of sampling router, including the need to buffer samples until their receipt is acknowledged. For this reason it may be desirable to use instead an unreliable transport, such as UDP. This in turn has ramifications for trajectory reconstruction, primarily the disambiguation of sample loss from packet loss. This issue will be the subject of further study.

Finally, the resource cost for storage and processing of trajectories arising from large networks may preclude the exclusive use of unaggregated trajectories. This motivates formulating an aggregation scheme for the storage of trajectories (at least for historical data) that minimizes information loss for the set of likely queries, even if general queries are less fully supported.

Acknowledgments. Thanks to Joel Gottlieb, Carsten Lund, and Fred True for assistance with usage and configuration data, Carsten Lund for help applying BRAVO, and Rick Greer for guidance on Daytona.

References

[1] Scalable Simulation Framework (SSF). http://www.ssfnet.org.

[2] D. O. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. A Framework for Internet Traffic Engineering. *Internet Draft (work in progress)*, November 2001.

[3] Cisco Corp. Netflow Services and Applications (white paper). August 1999.

[4] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Transactions on Networking*, 9(3):280–292, June 2001.

[5] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. NetScope: Traffic engineering for IP networks. *IEEE Network Magazine*, March 2000.

[6] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: Methodology and experience. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[7] Rick Greer. Daytona And The Fourth Generation Language Cymbal. In *Proc. ACM SIGMOD '99*, June 1999.

[8] John W. Stewart III. *BGP4*. Addison-Wesley, Reading, Mass., 1999.

[9] William Stallings. *SNMP, SNMP v2, SNMP v3, and RMON 1 and 2 (Third Edition)*. Addison-Wesley, Reading, Mass., 1999.

[10] Y. Vardi. Network Tomography. *Journal of the American Statistical Association*, March 1996.