Real-Time Detection of Hidden Traffic Patterns

Fang Hao Murali Kodialam T.V. Lakshman

Bell Labs
101 Crawfords Corner Road
Holmdel, NJ 07733
{fangh, muralik,lakshman }@bell-labs.com

Abstract

We address the problem of fast automatic identification of traffic patterns in core networks with high speed links carrying large numbers of flows. This problem has applications in detecting DoS attacks, traffic management, and network security. The typical measurement and identification objective is to determine flows that use up a disproportionate fraction of network resources. Several schemes have been devised to measure large flows efficiently assuming that the notion of what constitutes a flow is well defined a priori. However, there are many scenarios where traffic patterns are hidden in the sense that there is no clear knowledge of what exactly to look for and there is no natural a priori definition of flow. In this paper, we develop an effective scheme to identify and measure hidden traffic patterns. The approach is flexible enough to automatically identify interesting traffic patterns for further evaluation. The basic idea is to extend the runs based approach proposed in [1] to the case where flow definitions are not known a priori. A straightforward extension is both memory and processing intensive. We develop an efficient scheme that has good theoretical properties and does extremely well in practice.

1. Introduction

Accurate and fast flow measurement and characterization is an important component for network management, accounting, and traffic engineering. For instance, service providers may want to know which flows from which customers consume most of their network resources during any given time period, and adjust their provisioning and pricing accordingly. Network operators may want to constantly monitor the patterns of their network traffic in order to detect any suspicious changes in such patterns. A sudden increase in traffic to a particular destination may indicate a possible Denial of Service (DoS) attack.

Extensive research has recently been done on flow measurement [1, 3, 6] and packet sampling [4, 5, 9, 11, 12]. The proposed mechanisms typically use an explicit definition of traffic flow. A common definition is to characterize a flow by a 5-tuple in the IP packet header, including source IP address, source port, destination IP address, destination port, and protocol id. However, knowing what *type* of flow to capture or measure *before* actually conducting measurements is often not possible. Any combination of fields in the 5-tuple may constitute a flow with an "interesting" traffic pattern but this combination is not known a priori. In this sense, interesting traffic patterns are often hidden in traffic streams and efficient algorithms to uncover them in real-time do not exist to our knowledge.

As an example, an "interesting" flow to be observed may not be the 5-tuple flow, but flows defined by only destination address and port number. Network operators often do not know what flows to look for until they actually see statistics on various kinds of flows. Furthermore, measuring one particular type of flow may either lose or hide important information that can be derived by measuring other types of flows. For example, measuring only detailed 5-tuple flows may not reveal a possible on-going DoS attack since such attack may consist of many small 5-tuple flows. Similarly, measuring only aggregated flows based on destination address and port number may not reveal which source network uses most of the network bandwidth.

The work of Estan et al. [2] was the first to recognize this problem of hidden patterns and they proposed a traffic measurement algorithm that does not require an a priori flow definition [2]. They sift through traffic trace data and generate reports for multi-dimensional traffic clusters. Their approach can capture any flow with rate above a predefined threshold, regardless of flow dimensionality. This greatly improves usability and convenience for network operators. However, the approach requires scanning the trace multiple times and is essentially designed for off-line processing. The processing complexity and memory usage are not optimized for fast on-line measurement.

In general, we believe that a practical and real-time traffic measurement approach should have the following characteristics:
1.Accuracy: The approach should be able to measure flow rates with any desired level of accuracy.
2.Flexibility: The approach should not require a priori knowledge of flow definition. One only needs to specify the set of fields in an IP packet that may potentially be used to define a flow and the approach should

Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)
1092-1648/04 $ 20.00 IEEE

estimate any significant flows specified by any possible combinations of such fields.
3.On-line implementability: The processing should be simple and fast so as to support on-line measurement at high link speed. Measurement time required for deriving rate estimates should also be minimized.
4.Low Cost: The operation should have low implementation cost. For example, a solution that requires a large amount of SRAM is not considered practical.

Note that most existing mechanisms are aimed at satisfying requirements 1, 3, and 4 but not 2. The mechanism in [2] satisfies 1, 2, and 4 but not 3.

In this paper, we design a mechanism that satisfies all the above requirements. A naive solution is to simply extend RATE [1] or other per-flow estimation approaches such as [6] to measure flows with all possible dimensions. In other words, multiple estimators can be run in parallel, one for each flow dimension. It is easy to see that such solution suffers from scalability problem: it may drastically increase implementation cost (for hardware implementation) or reduce the line-speed that can be supported (for software implementation). When the number of fields that can constitute flows is 5, the complexity increases by a factor of 31 (i.e., $2^5 - 1$) for the multi-dimensional (flow definition unknown) case as opposed to the single-dimensional case where the flow definition is known. The problem becomes much worse if the IP prefixes are also considered when, for example, one wants to discover the big hitters between any two *networks*, instead of *hosts*. Given the common prefix length varies between 8 and 32 [1], one would need to track 5407 (i.e., $26^2 \times 2^3 - 1$) different types of flows. In addition, other IP header fields or some portions of the packet payload may also be included to allow finer grained statistics on traffic flows; any addition of such fields will further increase the number of estimators needed significantly.

We propose a solution that extends the RATE mechanism [1] by finding the longest match between two consecutive packets. Similar to RATE, the approach uses a packet register (or buffer) to store the last arrival packet and maintains a two-run count table (TCT). We show that this new approach can automatically detect all flows whose rate is above a given threshold without knowing flow dimension a priori. The approach is efficient in terms of accuracy, estimation time, memory cost, and operational overhead for each packet arrival. The minimum estimation time is a function of the specified accuracy level (same as RATE). The theoretical worst case operation complexity upon each packet arrival is $O(k^2)$, where k is the number of fields that can constitute flows; and simulations based on both synthetic data and real traffic data shows the average complexity is much less. We also show that the memory size required for the TCT table is insignificant in practice.

The rest of the paper is organized as follows: We first formally define the problem in Section 2 and summarize original RATE mechanism in Section 3. We then describe the new mechanism in Section 4, including both a naive implementation and a more efficient implementation. In Section 6 we present our experimental results based on both synthetic data and several real network traces. We finally conclude in Section 7.

1 http://isp-lists.isp-planet.com/isp-bgp/

2. Problem Definition

We consider a node in a network processing arrivals from multiple flows. An example is a router processing arrivals for multiple destination IP-addresses. Any field or combination of fields in the packet header can be defined to be a flow. We assume that the node is processing a large number of flows at any point in time. The objective of this paper is to design a *traffic rate estimator* to estimate the number of packets processed for each flow to any pre-specified level of accuracy. There are many different schemes that can be used to estimate the traffic rates for the different flows. These schemes can be compared based on several metrics. Three metrics that we consider in this paper are:
Sample Size: Sample size is defined to be the number of samples needed to achieve a desired level of accuracy. The larger the sample size, the longer is the time needed to estimate the traffic rates. If the traffic characteristics change over time, we would like the time scale needed to estimate the rates to be smaller than the time scale in which the traffic varies. Therefore, we would prefer a scheme with a small sampling size.
Memory Requirement: During the process of rate estimation, the estimation scheme keeps in memory traffic counts per-flow or a subset of the flows that is processed by the nodes. The memory requirement for the estimation scheme is proportional to the size of this subset of flows for which counts are maintained. We therefore use the *number of flows for which counts are maintained* as surrogate for the memory requirement. Keeping memory requirements low also leads to improved running time and ease of implementation. Keeping track of the arrivals for each flow, often referred to as per flow information, is clearly far more expensive and we argue that our scheme can be an effective substitute for accurate traffic estimation.
Average Processing Per Arrival: In order for the scheme to be implementable on-line, we have to minimize the processing per arrival. Such processing involves both operations of counting flows and updating flow-count table. The actual metrics are discussed in further details in later sections.

2.1. Notation

We assume that each arrival is tagged with a k-tuple integer. We denote a generic tag by $H = (h_1, h_2, \ldots, h_k)$ and the *tag* for arrival i by $H^i = (h_1^i, h_2^i, \ldots, h_k^i)$. We use the terms *header* and *tag* interchangeably in this paper. Let $p(H)$ denote the fraction of traffic that has tag H. We assume that the probability that an incoming arrival carries tag H is $p(h)$ and is independent of all other arrivals in the system. Unlike the single dimensional tag case where the definition of flows are natural, in k-dimensional case we need a couple of definitions before we can define a flow. The first is the notion of a *don't care* symbol. We use X to denote a don't care symbol in this paper. A don't care symbol in position j of a vector is used to indicate that we do not care

about component j of the vector. The don't care symbol is used to aggregate arrivals into flows. For example, the vector (a, X, X, X, X) is used to represent all vectors whose first component is a and is independent of all other components of the vector. Before we define a flow we first need this technical definition for the containment of two k-tags.

Definition 1 *Given two k-tags $G = (g_1, g_2, \ldots, g_k)$ and $H = (h_1, h_2, \ldots, h_k)$ (one or both of the tags can have X in some positions), $G \sqsupseteq H$ (G contains H) if and only if $g_j = h_j$ for all $j : g_j \neq X$*

Note that a tag always contains itself ($H \sqsupseteq H$); but the operator $\sqsupseteq$ is generally asymmetric. For example, if $G = (23, 34, X, X, 17)$ and $H = (23, 34, 42, X, 17)$ then $G \sqsupseteq H$ but not $H \sqsupseteq G$.

Using this definition of containment, we are now ready to define a flow.

Definition 2 *We define a* flow *to be a k-vector comprising of integers and don't care symbols X. Given a flow vector G, a packet with header H, is defined to belong to flow G if and only if $G \sqsupseteq H$.*

Note that a given packet with a k-tuple tag belongs to $2^k - 1$ flows, one corresponding to each subset (except the null set) of the original tag. For example a packet with tag $(12, 35, 18)$ belongs to flows $(12, X, X)$,$(X, 35, X)$,$(X, X, 18)$,$(12, 35, X)$,$(12, X, 18)$, $(X, 35, 18)$, and $(12, 35, 18)$. The objective of the sampling mechanism is to determine the fraction of traffic that belongs *all* flows to any pre-specified level of accuracy. In a networking context, we can think of the tag as the 5-tuple packet header comprising of (source-address, destination-address, source-port, destination-port, protocol-id). For example, packets that have source address a and destination address b, can be captured by the flow (a, X, b, X, X).

In this paper, we do not have to pre-specify which fields or combination of fields are of interest. The idea is for the mechanism to automatically identify interesting patterns of flow to perform network diagnostics and measurement. The objective of this paper is to design a sampling scheme that estimates the fraction of traffic belonging to each flow to any desired level of accuracy that we term the accuracy requirement.

Accuracy Requirement

We want to design a sampling scheme that for any given any flow whose fraction is p, determines $\hat{p}$ for p such that $\hat{p} \in \left(p - \frac{\beta}{2}, p + \frac{\beta}{2}\right)$ with probability greater than α. In other words, we are willing to tolerate an error of $\pm\frac{\beta}{2}$ with probability less than α. For example, the requirement on the sampling scheme can be the following: At the end of the sampling period, given any flow i determine p_i within an error of ± 0.0001 with a probability greater than 99.99%. This requirement translates to $\beta = 0.0002$ and $\alpha = 0.9999$. Throughout this paper, we use $\mathcal{N}[a, b]$ to represent a normal distribution with mean a and variance b. We use Z_α to denote the α percentile for the unit normal distribution. If $\alpha = 99.99\%$ then $Z_\alpha = 4.0$.

3. Summary of RATE

The techniques in this paper depend upon some of the results and analysis in RATE. Therefore, we briefly summarize RATE in this section. RATE can be viewed as a one dimensional version of the problem that we address in this paper. RATE can be applied effectively if we already know the combination of fields in the header that we are interested in tacking. This combination of fields that we are interested defines a flow in RATE. RATE is based on sampling only a subset of the arriving traffic at the node but it picks this subset carefully so that flows that send a larger proportion of the traffic are sampled more frequently. This is achieved by sampling two runs. For convenience of discussion, we assume that each flow is tagged with a unique integer. We assume that there are F flows in the network. The fraction of traffic belonging to flow i will be denoted by p_i and flow $i \in F$ is defined to have a two-run if two consecutive samples belong to flow i. The idea in RATE is to count the number of two runs that a flow generates and use this value to estimate the fraction of total traffic that belongs to that flow. Since small sources have very low probability of generating two-runs, the list of flows that are detected will be quite small. This leads to a memory efficient implementation. RATE detects and measures two-runs by maintaining the following information:

Two-Run Detecting Register R: This register needs to hold only one flow id, typically the last sample. If the current flow id is the same as that in the register, a two-run is detected, the Two-Run Count Table (described below) is updated and the run detecting register R is set to null. Otherwise, the flow id in R is replaced by the current lfow id.

Two-Run Count Table TCT: The Two-Run Count Table maintains counts for the number of two-runs for each flow that has a two-run. When a two-run is detected for a particular flow and if the flow is already in TCT then the two-run count for the table is incremented by one. If the flow for which a two-run has been detected is not in TCT, then this flow id is added to TCT and its count is initialized to one.

Note that the two-run detecting register is reset to null as soon as there are two arrivals consecutively from the same flow. This is done in order to make the analysis of the runs process simple. Resetting the register as soon as a two-run is detected, makes the point at which two-runs occur a regeneration point. We illustrate by means of an example the implication of this resetting. Let the sequence of flow ids be ...23,46,46,46,57.... Note that flow number 46 has a two run. As soon as a two run is detected, the two run counter is initialized to null. Therefore, the third 46 in the sequence will not be counted as two run though the second and third 46 can be viewed as a two run. If the sequence is however ...23,46,46,46,46,57.... then flow 46 would have had 2 two runs. This point is simple but is important in the multi-dimensional case. In general, if there are k arrivals in row for a given flow, then the flow will have $\lfloor \frac{k}{2} \rfloor$ two runs.

3.1. Estimation of Fraction

A point estimate $\hat{p}_i$ for the fraction of traffic sent by flow i is computed from the number of two runs $N_2(i,T)$ for flow i in T samples. In [1], it is shown that

$$\hat{p_i} = \frac{1}{2}\left(\frac{N_2(i,T)}{T} + \sqrt{4\frac{N_2(i,T)}{T} + \left(\frac{N_2(i,T)}{T}\right)^2}\right).$$

Therefore the estimate for the fraction for flow i can be computed in closed form. Note that, in general, there will be a large number of flows (especially small flows) which will not have two runs and hence the estimate for the fraction will be zero.

3.2. Estimating the Number of Samples

The following result that characterizes the variance of the point estimator is used to estimate the number of samples needed in order to meet the desired accuracy conditions for the estimator.

Theorem 1 *Let $N_2(i,t)$ be the number of two-runs for flow i in T samples and let*

$$g(x) = \frac{1}{2}\left(x + \sqrt{4x + x^2}\right).$$

Then

$$\sqrt{T}\left[g\left(\frac{N_2(i,T)}{T}\right) - p_i\right] \sim \mathcal{N}\left[0, \delta_i\right].$$

where

$$\delta_i = \frac{(1-p_i)\,(1+p_i)\,\left(1+3\,p_i+p_i^2\right)}{(2+p_i)^2}.$$

Note that δ_i represents the estimate for the variance and this takes on a maximum value of 0.345. Therefore the confidence interval by Equation will not be greater than

$$2\sqrt{\frac{Z_\alpha 0.345}{T}}.$$

We set this quantity to be less than β and solve for T to determine the length of time we have to sample to reach the objective that the α percentile confidence interval is less than β. The minimum sample size T is given by

$$T = \frac{4\,Z_\alpha^2\,0.345}{\beta^2} = \frac{1.38 Z_\alpha^2}{\beta^2}.$$

3.3. Memory Requirements

Recall that we use the number of entries in the TCT as a surrogate for the memory requirement. The basic idea in computing the memory requirement for the scheme is the observation that if the fraction of flow is sufficiently small, two runs for flow i come as a Poisson process with rate p_i^2. This fact is used to compute the worst case, expected table size. This result is used in conjunction with the number of samples in order to obtain the following result:

Theorem 2 *Let $L(T)$ represent the number of flows in TCT after T samples. Then*

$$E[L(T)] \leq 0.638\sqrt{T} = \frac{0.74 Z_\alpha}{\beta}$$

in the worst case.

Note that the expression for the sample size as well as the worst case memory requirement is independent of the number of flows, and is only dependent on the accuracy requirement.

4. Extending RATE for Vector Labels

We now analyze the flow estimation problem for the case where each packet has a k-dimensional vector label and each packet belongs in 2^k-1 flows (as defined in Section 2). In this section, we outline a straightforward extension of the RATE scheme to the vector label case. Although this implementation is very inefficient in terms of both memory and processing requirements, it provides the insights needed in order to develop a faster algorithm for the multi-dimensional flow counting problem. The naive implementation is outlined as follows. Each arrival belongs to $m = 2^k - 1$ flows. We define $S_1, S_2, \ldots, S_m$ subsets of $\{1, 2, \ldots n\}$. For example in the case of a 3-dimensional tag, the sets are $S_1 = \{1\}, S_2 = \{2\}, S_3 = \{3\}, S_4 = \{1,2\}, S_5 = \{1,3\}, S_6 = \{2,3\}, S_7 = \{1,2,3\}$. In RATE, there is one runs register R and one TCT . In the vector case we maintain $m = 2^k - 1$ runs registers $R_1, R_2, \ldots, R_m$ and m Two-Run Count Tables, $TCT(1), TCT(2), \ldots, TCT(m)$. We assume that the contents of all the registers are maintained in the form of k-flows. For example, the register R_4 corresponding to S_4 in the example above will have a 3-vector where the last component is a don't care. If the first arrival to the system is $H^1 = (32, 24, 56)$ then R_4 will have $(32, 24, X)$. In order to simplify notation, in the description of the algorithm below we use $R_4 = H^1$ to denote this operation of setting register R_4 with the values in H^1. Note that the don't cares in R_4 are maintained. We also assume that the contents of the TCT are also in this form. At the arrival of packet i with header $H^i = (h_1^i, h_2^i, \ldots, h_k^i)$, the following operations are performed:

For $j = 1, 2, \ldots, m$,
  If $R_j \sqsupseteq H$ then
    If $R_j \notin TCT(j)$ then add R_j to $TCT(j)$ with a count of one.
    If $R_j \in TCT(j)$ then increment the count of R_j by one.
    Set $R_j = \emptyset$.
  Else
    Set $R_j = H$

NAIVE EXTENSION OF RATE

We can bound the number of samples needed as well as the memory size for this algorithm. The expression of the number of samples needed for RATE is independent of the number of flows and is only dependent on the accuracy requirement. This leads to the following result:

Lemma 3 *The minimum sample size T for the naive extension of RATE is given by*

$$T = \frac{4\, Z_\alpha^2\, 0.345}{\beta^2} = \frac{1.38 Z_\alpha^2}{\beta^2}.$$

In order to bound the amount of memory required for the naive extension of RATE, note that the implementation is exactly as if there are $2^k - 1$ independent TCT in RATE. From Theorem 2, we directly obtain the following result:

Lemma 4 *The expected amount of memory required for a naive extension of RATE with confidence interval width β and the error probability α is given by*

$$E[L(T)] = \frac{0.74(2^k - 1)Z_\alpha}{\beta}.$$

There is obviously an increase in memory requirement by a factor of $2^k - 1$ over RATE. More importantly, the processing complexity upon each packet arrival increases from $O(k)$ in RATE (matching each field) to $O(k \cdot 2^k)$. This processing overhead makes the scheme not practical even for small values of k. In the next section, we give an alternate implementation of the algorithm that has a processing requirement of $O(k^2)$ and also leads to a drastic reduction in memory requirement in practice. We are now ready to describe an efficient implementation of a runs based detector that reduces the processing overhead of RATE. This algorithm that we call Traffic Pattern Detector (TRAPPED) is described in the next section.

5. Traffic Pattern Detector (TRAPPED)

The basic idea in the Traffic Pattern Detector (TRAPPED) is to decrease the real-time processing time while increasing the processing requirement at the time of querying after the sampling is done. We maintain only one k-dimensional runs register R and one TCT and we count runs for the longest string only. For example if the first two headers are $(23, 12, 34, 45, 67)$ and $(23, 12, 15, 45, 6)$, instead of counting this as a two run for the seven flows that are subsets of $(23, 12, X, 45, X)$, we just maintain it as a two-run for $(23, 12, X, 45, X)$. This saves both memory and processing time, since we only look at the longest string and not every subset of it. For each flow $G \in TCT$ there will be an associated two run counter $N_2(G)$. At the end of the sampling process, if we are given a flow $F = (f_1, f_2, \ldots f_k)$, the following routine is used to determine the number of two runs for that flow.

Set $COUNT = 0$.
For each flow $G \in TCT$ do
  If $F \sqsupseteq G$ then
    $COUNT = COUNT + N_2(G)$.

COUNTING TWO RUNS FOR FLOW F

Note that unlike the naive extension of RATE, where the count for all flows are readily available, in the more efficient implementation, the count for a given flow has to be generated from the TCT. Therefore, the processing requirement at the time of querying is higher. In other words, TRAPPED *decreases real-time processing and increases query time processing* in order to achieve this efficiency in real time speed as well as memory. The scheme for maintaining the longest string though simple is non-trivial.

5.1. Efficient Tracking of Two Runs

For the efficient tracking of two runs in the k-dimensional case, we maintain a k dimensional run vector R and a corresponding k-dimensional counter C. We use $R^i = (r_1^i, r_2^i, \ldots r_k^i)$, and $C^i = (c_1^i, c_2^i, \ldots, c_k^i)$ to denote the run vector and the counter after arrival i. These vectors are updated as follows:

At arrival i set

$$c_j^i = \begin{cases} 1 & \text{if } h_j^i \neq r_j^{i-1} \\ c_j^{i-1} + 1 & \text{if } h_j^i = r_j^{i-1} \end{cases}$$

$r_j^i = h_j^i \;\; 1 \leq j \leq k.$

UPDATING R^i AND C^i.

We now illustrate via an example how these counters are used and then formally describe the process of updating the TCT.

Let $R^{i-1} = (28, 18, 52, 16, 22)$ and the corresponding $C^{i-1} = (1, 3, 3, 2, 2)$. Let $H^i = (28, 34, 52, 16, 22)$. Then $R^i = (28, 34, 52, 16, 22)$ and the corresponding $C^i = (2, 1, 4, 3, 3)$. We now have to update the TCT. We ignore fields that have a run length of 1 in all the analysis. Unlike the one dimensional tag, note that different fields might have different run lengths at any given point. We start with the entry with the smallest run length, in this case 2. Note that $(28, X, 52, 16, 22)$ represents the longest string (flow) that as a two run. Consider the flow $F = (X, X, X, 16, 22)$. Note that arrival i does not create a two run for F since the run length for F is only 3. However, by incrementing the counter for $(28, X, 52, 16, 22)$ we have incremented the two-run count for F. This has to be corrected. More generally, whenever the count is an odd number then there is no two run. In order to keep the counts accurate, we decrement TCT counter corresponding to strings with odd number of two-runs, in this case the string $(X, X, 52, 16, 22)$. The net effect of this increment and decrement is that running COUNT TWO RUNS with F will not result in an increment in count for F. However, this will also be the case for the string $(X, X, 52, X, X)$ which has had a 4-run. Therefore, we increment the TCT counter for $(X, X, 52, X, X)$.

In summary we add three flows to the TCT: (1) Increment $(28, X, 52, 16, 22)$ by one. (2) Decrement $(X, X, 52, 16, 22)$ by one. (3) Increment $(X, X, 52, X, X)$ by one. It is easy to verify that for any flow, the net count will be correct. Note that such alternating increment and decrement are necessary only if there are

interspersed odd and even numbers in C^i. In the above example, if we had $C^i = (2, 1, 4, 6, 6)$, then we would only need to increment $(28, X, 52, 16, 22)$ by one.

In order to formalize this algorithm we need the following definition.

Definition 3 *Given a R and C vector and an integer value $\gamma > 1$, we define the γ level vector to be a k-vector $V(R, C, \gamma)$ such that*

$$V_j(R, C, \gamma) = \begin{cases} r_j & \text{if } c_j \geq \gamma \\ X & \text{otherwise} \end{cases}$$

We now illustrate the definition with an example: Let $R^{i-1} = (28, 18, 52, 16, 22)$ and the corresponding $C^{i-1} = (1, 3, 3, 2, 2)$. Let $H^i = (28, 34, 52, 16, 22)$. Then $R^i = (28, 34, 52, 16, 22)$ and the corresponding $C^i = (2, 1, 4, 3, 3)$. Note that $V(R^i, C^i, 4) = (X, X, 52, X, X)$,$V(R^i, C^i, 3) = (X, X, 52, 16, 22)$ and $V(R^i, C^i, 2) = (28, X, 52, 16, 22)$. In addition to this we assume that we have a function $PARITY()$ that takes as input an integer and outputs whether the integer is odd or even. For example $PARITY(4) = EVEN$ and $PARITY(3) = ODD$. The following is a description of the procedure followed at each arrival:

Update R^i and C^i.
Compute the largest value l, in C^i.
Set $CURPAR = ODD$.
For $\delta = 2, 3, \ldots, l$ do
  If ($\delta \in C^i$ and $PARITY(\delta) \neq CURPAR$)
    If $PARITY(\delta) = EVEN$ Increment $V(R^i, C^i, \delta)$
    If $PARITY(\delta) = ODD$ Decrement $V(R^i, C^i, \delta)$
    $\delta \leftarrow \delta + 1$
    $CURPAR = PARITY(\delta)$
  $\delta \leftarrow \delta + 1$

ALGORITHM TRAPPED

In the description of TRAPPED note that when we say increment or decrement some flow in TCT, we increment or decrement the counter corresponding to that flow. If the flow is not currently in the TCT, then we insert the flow into the TCT. Note that if a flow is not in the TCT and we decrement the counter, then it is equivalent to inserting the flow in the TCT with a count of -1. Also note that the main loop just needs to iterate over the different components in C^i and not through each value of δ between 2 and l. This is done in the description for clarity. We illustrate the steps of the algorithm using an example: As in the previous example, let $R^{i-1} = (28, 18, 52, 16, 22)$ and the corresponding $C^{i-1} = (1, 3, 3, 2, 2)$. Let $H^i = (28, 34, 52, 16, 22)$. Then $R^i = (28, 34, 52, 16, 22)$ and the corresponding $C^i = (2, 1, 4, 3, 3)$. Note that $V(R^i, C^i, 4) = (X, X, 52, X, X)$,$V(R^i, C^i, 3) = (X, X, 52, 16, 22)$ and $V(R^i, C^i, 2) = (28, X, 52, 16, 22)$. As per the algorithm, the counts for $V(r^i, c^i, 4) = (X, X, 52, X, X)$ and $V(r^i, c^i, 2) = (28, X, 52, 16, 22)$ are incremented by one. (We assume that these two entries are already in the TCT, if not we insert these vectors in the TCT with a count of one). The count for $V(r^i, c^i, 3) = (X, X, 52, 16, 22)$ is decremented by one. If this string is not already present in the TCT, then the vector is inserted into the TCT with an initial value of -1.

Theorem 5 *Algorithm TRAPPED computes the entries to the TCT accurately with a running time of $O(k^2)$ every time there is some component of C^i that is greater than one.*

Proof (outline)
The algorithm starts with $\delta = 2$. All the fields that have greater than one therefore have had a two run. The complication is that some of these two runs have already been accounted for. These are the two runs corresponding to fields that have an odd counter. This is due to the fact that at the previous arrival, these fields would have had a two-run. From the definition of a two run, three arrivals in a row constitutes only one two run. Therefore, we have to decrement the fields that correspond to these odd numbers. Just as addition to the TCT is the longest string, decrementing is also done on the basis of longest strings. This process of alternating increments and decrements for even and odd numbers is repeated until we reached the highest value in the run length counter. Note that at each iteration in TRAPPED we have to determine the string corresponding to the current value of δ. This takes $O(k)$ time. There are at most k iterations giving a running time of $O(k^2)$.

Therefore the TRAPPED reduces the running time from $O(k \cdot 2^k)$ to $O(k^2)$.

5.2. Dealing with Prefixes

The IP source and destination addresses can be thought of as having multiple sub-fields and a match between two source addresses is based on the longest prefix match. We denote prefixes in square brackets. As an example, consider a field in the 5 dimensional header for a packet where the first field has 3 sub-fields. Packets $([32, 45, 27], 89, 21, 12, 56)$ and $([32, 45, 62], 89, 45, 12, 56)$ have longest match of $([32, 45, X], 89, X, 12, 56)$. While packets $([32, 45, 27], 89, 21, 12, 56)$ and $([49, 45, 27], 89, 45, 12, 56)$ have longest match of $(X, 89, X, 12, 56)$. Commonality of subfields is defined up to the first place where the two subfields differ. As far as TRAPPED is concerned, the presence of subfields does not alter the algorithm. Of course, the length of the runs array R and the counter C will be higher to accommodate the subfields. The algorithm does not change significantly. The presence of prefixes (which use Longest Prefix Matching) changes the update algorithm for R^i and C^i. Assume that field a has b subfields where there is a longest prefix match in the subfields. We represent the run vector for this field a as r^i_{aj} where $j = 1, 2, \ldots b$. The packet correspondingly has a header h^i_{aj} where $j = 1, 2, \ldots b$. We now give the run update algorithm for field a. (The update algorithm for the fields that do not have prefix subfields is shown in Section 5.1).

At arrival i set

$$c^i_{a1} = \begin{cases} 1 & \text{if } h^i_{a1} \neq r^{i-1}_{a1} \\ c^{i-1}_{a1} + 1 & \text{if } h^i_{a1} = r^{i-1}_{a1} \end{cases}$$

For $j = 2, 3, \ldots, b$,

$$c^i_{aj} = \begin{cases} c^{i-1}_{aj} + 1 & \text{if } h^i_{aj} = r^{i-1}_{aj} \text{ and } c^{i-1}_{a(j-1)} < c^i_{a(j-1)} \\ 1 & \text{otherwise} \end{cases}$$

$r^i_{aj} = h^i_{aj} \;\; 1 \leq j \leq b.$

UPDATING R^i AND C^i FOR PREFIXES.

Note that unlike the case where the fields are independent, in the case of prefixes, the update for a sub-field depends upon whether all the subfields up to that point are matched. Once the values of R^i and C^i are computed, the rest of the algorithm is the same as described in Section 5.1.

5.3. Some Remarks

1. Throughout the analysis, we assumed that the packet headers form an independent process. In practice, if the packet headers can be correlated. In this case, it is easy to overcome this problem using multiple run and count registers. Packets are compared in a round-robin fashion across the different run registers. This implies that we will not compare the current packet header with the previous packet header but against some header in the past. The larger the number of register, the more is the potential for "washing out memory". Note that the analysis in RATE and in this paper do not assume that we compare two consecutive packets to determine two-runs. Therefore the number of samples as well as memory requirement estimates remain the same.
2. Theoretically, both TRAPPED and naive extension of RATE have the same worst case memory requirements. However, as we illustrate in next section, we find through simulations that the actual memory requirement of TRAPPED is at least one order of magnitude smaller than naive implementation. The difference increases with increase in flow dimension.
3. In TRAPPED, we need to derive the actual two-run count for each flow by running COUNTING TWO RUNS algorithm. The complexity of deriving actual two-run counts for all flows is $O(L^2(T))$, where $L(T)$ is the TCT table length. In practice, the actual measurement and COUNTING TWO RUNS algorithm can run in pipeline, so that statistics for all flows at each time window can be continuously generated in real time.

6. Experimental Results

In this section, we present our performance evaluation results for TRAPPED. We intend to measure the accuracy, estimation time, memory cost, and operational complexity of the mechanism. In the following, we first describe the experimental setup and list the metrics that are used, and then explain the results based on both synthetic traces and real IP data traffic traces from NLANR[2].

6.1. Simulation setup

In the simulation, we first process each arrival packet according to the TRAPPED algorithm specified in Section 5. TCT tables are updated when two-runs based on any fields or prefixes are detected. This process continues until the end of the sampling period, which is calculated as in Lemma 3. As the final step, we invoke COUNTING TWO RUNS algorithm to traverse through TCT table to generate the actual two-run count for all flows captured.

Multiple packet registers are used in simulation to minimize the auto-correlation within the trace. The number of registers depends on the auto-correlation structure of the trace. Through simulation, we find that number of registers on the order of a few hundred to a thousand usually suffice. We choose to use 1000 registers in simulation.

We use both randomly generated synthetic traffic data and real IP packet data traces in our simulations. In both cases, we use 5-tuple from the header to identify a packet.

Synthetic data. Synthetic data are generated as follows: we first produce a list of 150 flows; each flow is assigned a rate[3] that is selected from a uniform distribution between [0.003, 0.005]. Each field within 5-tuple of a flow is marked as *don't care* (i.e., it can be of any value) with probability p_{dc}. The fields that are not marked as *don't care* are filled in with specific patterns described below. Source and destination ports are filled in with a random integer selected between [0,65535]. Protocol id is filled in with a random integer selected between [0,255]. For source and destination addresses, first a prefix length is selected from a uniform distribution between [8,32], and then an IP prefix is generated randomly according to the prefix length. The portion of the IP address following the prefix are again marked as *don't care*. For example, a flow in the list may look like this: {srcIP=10.0.X.X, srcPort=X, destIP=X, destPort=X, prot=30}, where "X" means *don't care*. In the above process, we make sure that at least one field is not marked as *don't care* for each flow; and no two flows are identical.

In order to generate flows according to their specified rate, we partition the rate range [0,1] into 151 segments, where the length of each segment equals to the rate of each corresponding flow. Therefore each of 150 flows is mapped to a different segment. The last remaining segment represents packets with all fields marked as *don't care*. In each step of the simulation, a random number is selected from a uniform distribution between [0,1]. Once we select a flow based on which segment this random number falls in, we then generate a packet by filling in its 5-tuple according to

2 http://pma.nlanr.net

3 Actual flow rate in terms of number of packets per second can be derived from the total traffic rate and the proportion of each flow in the total traffic. Hence, we only estimate the proportion of each flow in total traffic, and call this proportion "flow rate" for convenience. Similarly, the sampling time is measured in terms of number of packets sampled in our experiments.

the chosen flow pattern. Portions that are marked as *don't care* will be filled in randomly according to the value range of that particular portion.

IP trace data from NLANR. We use seven sets of IP trace data that have been collected between April 1, 2004 and May 5, 2004 on various OC-3 and OC-12 links, available from NLANR. Table 1 shows a list of the traces. Each trace contains a sequence of IP headers along with their arrival time. Note that source and destination IP addresses are anonymized; and IP prefix structures are not preserved during transformation. Hence we only use the traces to partially validate TRAPPED under the scenario where IP prefixes are not identified.

Trace name	Date	Link type	Length	# packets
BWY1	4/1/04	OC-3	90 sec.	2.28 M
BWY2	4/25/04	OC-3	90 sec.	2.59 M
BWY3	5/5/04	OC-3	91 sec.	2.50 M
COS	4/25/04	OC-3	90 sec.	2.11 M
FRG	5/5/04	OC-3	90 sec.	3.62 M
MRA1	4/25/04	OC-12	90 sec.	6.28 M
MRA2	5/5/04	OC-12	90 sec.	6.23 M

Table 1: NLANR IP trace data

6.2. Metrics

We intend to measure accuracy, estimation time, memory cost, and operational complexity of TRAPPED in our experiments.

To measure accuracy, we compare *estimated flow rate* with *actual flow rate*. Estimated flow rate is derived from two-run count by applying the same formula as in RATE (see Section 3.1). Actual flow rate is measured by directly counting packets for each individual flow, which is in turn divided by total number of sampled packets. It is not easy to measure actual rate for all flows because both the number of dimensions and the number of individual flows in each dimension can be very large. We limit the set of flows to count as follows: for synthetic data, we only measure the actual rate of "big" flows that are in the list; for NLANR trace, we first apply TRAPPED to the trace to get a list of all flows that generate two-runs, and then go through the trace again to directly count all such flows.

Sampling time are calculated based on Lemma 3, same as that of the original RATE scheme [1]. We choose confidence interval (β) and error probability (α) as 0.002 and 99.95% for synthetic data, and 0.004 and 99.75% for NLANR traces, respectively. Consequently, the target sampling time for synthetic data and NLANR traces are 12.4 million packets and 3.1 million packets, respectively. Note that some of the NLANR traces are slightly shorter than the target sampling time, and hence the rate estimation accuracy for such traces are also slightly affected.

Several factors contribute to memory cost: packet register (R^i) and associated counters for length of runs (C^i), and TCT table. Here we focus on TCT table size since it may dynamically change during estimation, while the size of packet register and length of runs counters are fixed. TCT table size is measured in terms of number of flow entries in the table.

TRAPPED consists of the following basic operations: (1) matching upon each arrival; (2) counting length of runs for each field when some fields of two arrival packets match; and (3) generating two-runs and updating TCT table when necessary. Matching and length of run calculation have complexity of $O(k)$, where k is length of all fields. These are relatively simple operations that can be implemented in hardware to support high line speed. Generating two-runs are updating TCT table are likely to be more time consuming, and hence are the focus in our evaluation. We use *average number of TCT accesses per arrival* as an indicator for this part of operational cost.

6.3. Estimation accuracy

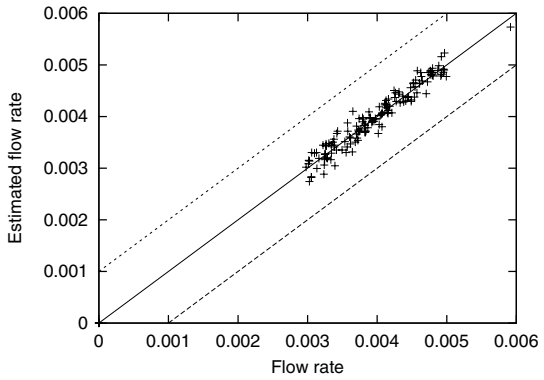

Figure 1: Flow rate estimation for synthetic trace, $p_{dc} = 0.5$

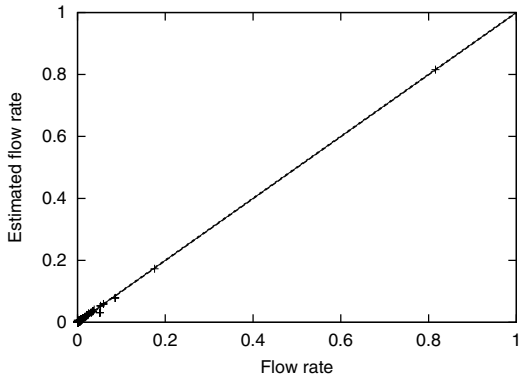

Figure 2: Flow rate estimation for NLANR trace (MRA1): full view

We compare the estimated flow rate with actual flow rate for synthetic data and NLANR traces. Figure 1 shows the results for synthetic data where the probability for generating *don't care* fields p_{dc} is 0.5. Results for other p_{dc} values are similar and not shown here. We observe that all data points fall well within the boundary between $y = x \pm 0.001$, indicating that estimated rate is very accurate. Figure 2 shows the results for NLANR trace MRA1. We observe that almost all data points are closely spread around $y = x$, similar to synthetic data case. Note that the figure includes about 1500 flows that are captured by TRAPPED and also has

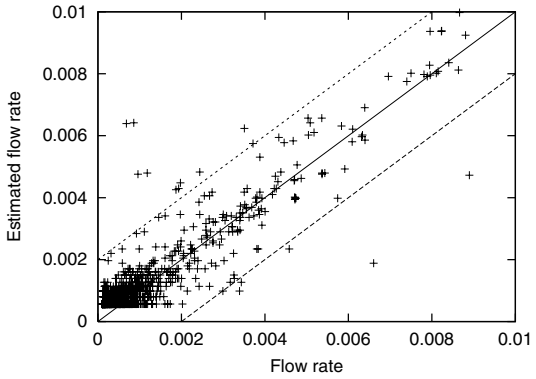

Figure 3: Flow rate estimation for NLANR trace (MRA1): zoom-in view

three lines including $y = x$ and $y = x \pm 0.002$, although they are not much recognizable due to the scale. Figure 3 shows the same data as that in Figure 2 with enlarged view between rate of [0, 0.01], where most data points fall in. Again, we can verify that most data points are within the $y = x \pm 0.002$ boundary. Note also from Figure 2 that there are a few flows with extremely high rates, ranging up to about 80%. We list a few of such large flows in Table 2. We find that TCP traffic (prot=6) consists of about 82% of the total traffic. The remaining traffic are mostly UDP (prot=17), roughly 18% of the total. About 8% and 6% are traffic to port 8000 (possibly port forwarding traffic to proxy servers) and port 80 (web traffic), respectively. Most traffic destined to port 8000 is UDP (prot=17). Almost all traffic destined to port 80 is TCP (prot=6). We may also find that host "10.0.0.9" sends a significant amount of traffic (3%) to host "10.0.0.10", which are all UDP (prot=17).

{srcIP,srcPort,dstIP,dstPort,prot}	Rate	Est. rate
{x,x,x,x,6}	0.8159	0.8156
{x,x,x,x,17}	0.1755	0.1728
{x,x,x,8000,x}	0.0859	0.0789
{x,x,x,8000,17}	0.0854	0.0784
{x,x,x,80,x}	0.0590	0.0592
{x,x,x,80,6}	0.0590	0.0592
{x,80,x,x,x}	0.0523	0.0528
{x,80,x,x,6}	0.0523	0.0528
{x,1148,x,x,x}	0.0511	0.0312
{10.0.0.9,8000,10.0.0.10,8000,17}	0.0343	0.0353

Table 2: Sample of large flows in NLANR MRA1 trace

6.4. Memory cost

Figure 4 shows the two-run count table size for synthetic data under different values of p_{dc}. Results for both TRAPPED and naive RATE approaches are presented for comparison. Note that the vertical axis is in log scale. We find that number of flows captured in TCT for TRAPPED is around 15 K, several orders of magnitude smaller than that of naive approach (ranging from 150 K to 10 M). We also observe that TCT size for naive approach increases almost exponentially as p_{dc} decreases. This is because decrease in

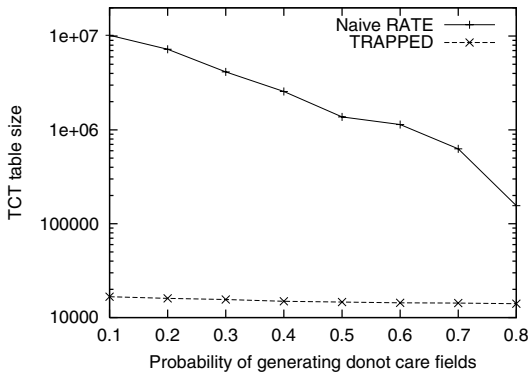

Figure 4: Two-run count table size for synthetic data

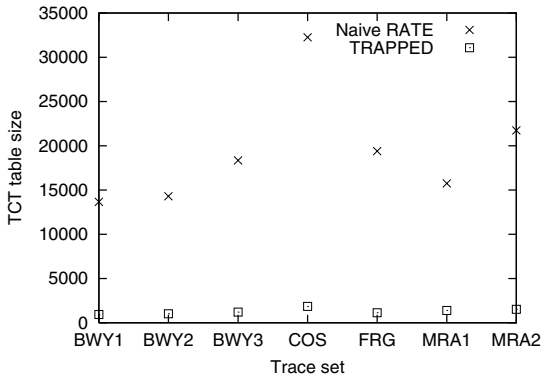

Figure 5: Two-run count table size for NLANR traces

p_{dc} leads to increase in average flow length, i.e., average number of valid fields in a flow. Longer flows implies larger dimension, which cause more flow entries to be generated in TCT for naive RATE approach. This is not an issue for TRAPPED since it typically only generates one entry for the longest flow (largest dimension) when match occurs.

Figure 5 shows the two-run count table size for NLANR traces under both TRAPPED and naive RATE approaches. We observe that difference between TRAPPED and naive RATE is about one order of magnitude. Note that max flow dimensions are 5407 and 31 for scenarios with and without prefix matching, respectively. We expect the TCT size difference between TRAPPED and naive RATE to be even more significant when prefixes are considered. Such dimension difference also explains why the TCT size for NLANR traces (about 900 to 1800) are smaller than that of synthetic traces (about 15,000).

6.5. Operational overhead

Figure 6 shows the average number of TCT accesses per packet arrival for synthetic data for both TRAPPED and naive RATE. We find that average number of accesses for TRAPPED is about 0.01, much smaller than that of naive RATE (ranging from 0.1 to 2.0). However, the average number of accesses for TRAPPED under NLANR is quite significant (Figure 7), about 0.4 to 0.5; it is only slightly better than naive RATE. The major reason is that since most traffic on the link are either TCP or UDP, the corresponding flows {X,X,X,X,prot=6} and {X,X,X,X,prot=17} have

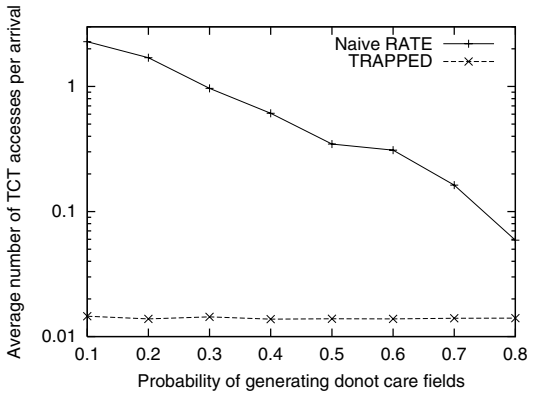

Figure 6: Average number of TCT accesses per arrival for synthetic data

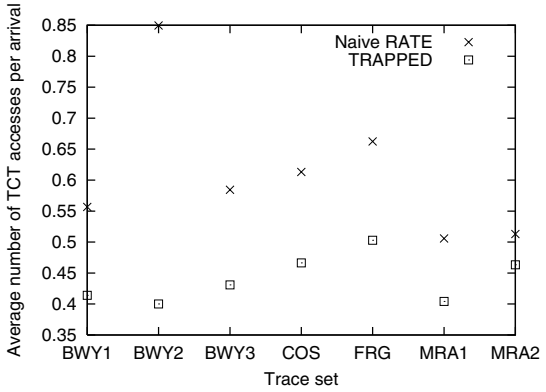

Figure 7: Average number of TCT accesses per arrival for NLANR trace

extremely high rate (0.82 and 0.18), and hence they generate a large number of two-runs and significantly increase number of TCT accesses. Note that the presence of such extremely heavy hitters does not affect TCT table size, since it does not increase the number of different flows. We also observe that the average number of accesses without maintaining such two-runs is one or two orders of magnitude smaller.

We can address this problem by adding a small cache in front of the TCT table. When a two-run count is generated, we first look up the cache to see if this flow already exists. If so, modify the two-run count of the existing entry. When the cache is full, flows with lower two-run count should be flushed out to TCT table first. Flows with same two-run count follow first in first out order. Since there are just very few such extremely heavy hitters, they will most likely remain staying in the cache. Therefore most counting for such flows can be done in the cache, without access TCT table. By this design, we can implement the TCT table by using slower and less expensive DRAM, and implement the small cache using faster and more expensive SRAM or TCAM.

In summary, we find that the simulation results indicate that TRAPPED can accurately estimate rate of multi-dimensional flows without knowing such flows or dimensions a priori. Both memory size and operational overhead are fairly small, which makes it promising to be implemented to support measurement on high-speed network links.

7. Conclusions

In this paper, we considered the problem of detecting hidden patterns in network traffic. The patterns are considered hidden because the real time uncovering of these patterns in the traffic stream requires a priori knowledge regarding the set of "interesting" flows. The main contribution of the paper is the development of a runs based detection algorithm TRAPPED for detecting hidden traffic patterns. We showed both theoretically as well as experimentally that the algorithm is very efficient in terms of the amount of processing per packet as well as the amount of time needed to do the detection. The algorithm is simple and can be implemented in real time and can be used in a wide variety of applications including traffic monitoring, anomaly detection and network security.

References

[1] Kodialam, M., Lakshman, T. V., and Mohanty, S., "Runs bAsed Traffic Estimator (RATE): A simple, Memory Efficient Scheme for Per-Flow Rate Estimation", *Proceedings of INFOCOM'2004*.

[2] Estan, C., Savage, S., and Varghese, G., "Automatically Inferring Patterns of Resource Consumption in Network Traffic", *Proceedings of ACM SIGCOMM 2003*.

[3] Kumar, A., Xu, J., Wang J., Spatschek, O., and Li L., "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement", *Proceedings of ACM INFOCOM 2004*.

[4] Duffield, N, Lund, C., and Thorup, M., "Charging from Sampled Network Usage", *SIGCOMM Internet Workshop 2001*.

[5] Duffield, N, and Grossglauser, M., "Trajectory Sampling for Direct Traffic Observation", *Proceedings of ACM SIGCOMM 2000*.

[6] Estan, C, and Varghese, G., "New Directions in Traffic Measurement and Accounting", *Proceedings of ACM SIGCOMM 2002*.

[7] Fang, W, and Peterson, L., "Inter-AS Traffic Patterns and their Implications", *Proceedings of IEEE GLOBECOM 1999*.

[8] Feldmann, A. et al., "Deriving Traffic Demands for Operational IP Networks: Methodology and Experience",*Proceedings of ACM SIGCOMM'2000*.

[9] Feng, W. et al., "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness", *Proceedings of INFOCOM'2001*.

[10] Heyman, D.P., and Sobel, M.J., *Stochastic Models in Operations Research, Vol 1.*, McGraw Hill, 1982.

[11] T. J. Ott, T. V. Lakshman, and L. H. Wong, "SRED: Stabilized RED", *Proceedings of INFOCOM'99*, pp. 1346-1355, Mar. 1999.

[12] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation", *Proceedings of INFOCOM'99*, pp. 2:942-951, Mar. 2000.