New Directions in Traffic Measurement and Accounting

Cristian Estan and George Varghese

Abstract— **Accurate network traffic measurement is required for accounting, bandwidth provisioning, and detecting DOS attacks. However, keeping a counter to measure the traffic sent by each of a million concurrent flows is too expensive (using SRAM) or slow (using DRAM). The current state-of-the-art (e.g., Cisco NetFlow) methods which count periodically sampled packets are slow, inaccurate, and memory-intensive. Our paper introduces a paradigm shift by concentrating on the problem of measuring only "heavy" flows — i.e., flows whose traffic is above some threshold such as 1% of the link. After showing that a number of simple solutions based on cached counters and classical sampling do not work, we describe two novel and scalable schemes for this purpose which take a constant number of memory references per packet and use a small amount of memory. Further, unlike NetFlow estimates, we have provable bounds on the accuracy of measured rates and the probability of false negatives. We also propose a new form of accounting called *threshold accounting* in which only flows above threshold are charged by usage while the rest are charged a fixed fee. Threshold accounting generalizes the familiar notions of usage-based and duration based pricing.**

I. Introduction

If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants. — Van Jacobson, End-to-end Research meeting, June 2000.

Measuring and monitoring network traffic is required to manage today's complex Internet backbones [3], [18]. Such measurement information is essential for short-term monitoring (e.g., detecting hot spots and denial-of-service attacks), longer term traffic engineering (e.g., rerouting traffic and upgrading selected links), and accounting (e.g., to support usage based pricing). The standard approach advocated by the Real-Time Flow Measurement (RTFM) [20] Working Group of the IETF is to instrument routers to add flow meters at either all or selected input links. Today's routers offer tools such as NetFlow [1] that give flow level information about traffic.

The main problem with the flow measurement approach is its lack of *scalability.* Measurements on MCI traces as early as 1997 [19] showed over 250,000 concurrent flows. More recent measurements in [7] using a variety of traces shows the number of flows between end host pairs in a one hour period to be as high as 1.7 million (Fix-West) and 0.8 million (MCI). Even with aggregation, the number of flows in 1 hour in the Fix-West used by [7] was as large as 0.5 million. Thus we believe that the flow measurement approach, with or without real time aggregation, *does not scale with the number of flows and increasing link speeds.* Keeping traffic counters for each such flow, as is done by NetFlow, is infeasible at high speeds. Cisco recommends the use of sampling at speeds above OC-3: only the sampled packets result in updates to the flow cache that keeps the per flow state. But this sampling has problems of its own since it affects the accuracy of the measurement data.

Despite the large number of flows, a common observation found in many measurement studies (e.g., [3], [7]) is that a small percentage of flows accounts for a large percentage of the traffic. [7] shows that the top 9% of the flows between AS pairs accounts for 90% of the traffic in bytes between all AS pairs. [12] shows even stronger results with 1% of flows accounting for 80% of traffic. These observations are used in [15], [12], [7], [3] to suggest that scalable differentiated services could be achieved by providing selective treatment only to a small number of flows that are "heavy hitters" [3].

However, how are such heavy hitters identified? This is the central question addressed by this paper. We present two algorithms that identify the heavy hitters using a small amount of state. Furthermore, we have strict, low worst case bounds on the amount of per packet processing, making our algorithms suitable for use in high speed routers.

II. Our solution

Our solution relies on a simple idea: identifying the largest flows and counting *all* their packets once they are identified. Counting all packets provides more accurate results than relying on samples but requires updating the flow memory on every packet, which requires that the flow

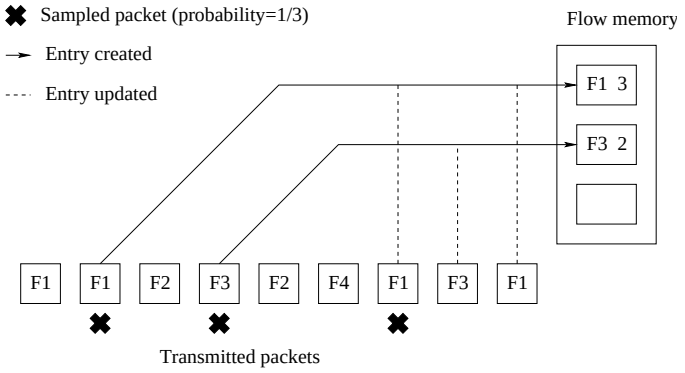

Fig. 1. After an entry is created for a flow the counter is updated for all its packets

memory move from slow DRAM (as in NetFlow) to fast SRAM. However, since SRAM is much more expensive than DRAM, the flow memory must be small. The crucial mechanism required to make this idea work is an algorithm that identifies large flows accurately (negligible rate of false negatives) and yet only tracks a minimal number of flows that are not large (small number of false positives).

Two simple approaches to identifying large flows suggest themselves immediately. The first is to use an associative memory that contains a small number of flow IDs tagged with counters. When a packet arrives with a flowID not in the flow memory, we make place for the new flow by removing the flow with the smallest measured traffic (i.e., smallest counter). It is easy, however, to provide counterexamples where a large flow is not measured because it keeps being expelled from the flow memory before its counter becomes large enough. A second approach is to use classical random sampling. Random sampling provably identifies large flows but using the sample size as a rate estimator has a large variance, and can hardly be used as a basis for accounting.

A. Sampled Counting

Our first method differs from classical sampling because we use random sampling *only to determine whether a flow will be examined more thoroughly*. Once a flow has been sampled we will count *all* its packets unlike Sampled NetFlow. Intuitively, random sampling identifies a large flow F quickly, and the accurate measurement provides an accurate rate for F once it has been sampled. Figure 1 gives an example of how our sampling solution operates.

Besides not providing accurate rates after sampling, Sampled NetFlow samples periodically and without taking into account packet sizes. To provide provable bounds, we sample packets randomly[1] such that each byte has equal probability p of causing the flow to be added to the flow memory. The sampling probability for a packet of size L becomes $p_L = 1 - (1-p)^L$. This can be looked up in a precomputed table or approximated by $p_L = p*L$. Finally, we choose p such that with overwhelming high probability, no sampled flow will have to be evicted from the flow memory.

The following example illustrates the method and the analysis more concretely. Suppose we wish to measure the traffic sent by all the flows that take over 1% of the link bandwidth in some period t. There are at most 100 such flows that take over 1%. Instead of making our flow memory have just 100 locations, we will allow oversampling by a factor of 100 and keep 10,000 locations in our SRAM. Each memory location contains space for a flow ID and a counter. We wish to sample each byte with probability p such that the average number of samples is 10,000. Thus if M bytes can be transmitted in S seconds, $p = 10,000/M$.

The algorithm is as follows. The flow memory is a CAM or a hash table. At the start of the time period, the flow memory is empty. Subsequently, whenever a packet of length L representing flow F arrives, we decide to sample the packet with probability $p_L = 1 - (1-p)^L$, where $p = 10,000/M$. If the packet is sampled, and flow F is not already in the flow memory, we add an entry to the flow memory with F and a counter value of 0. Next, *regardless of whether the packet was sampled or not,* we lookup the counter for F and increment it by L. At the end of S seconds, we output all the flows whose counters are over say $M/200$ together with their counters.

Here is the analysis. Consider a flow F that takes more than 1% of the traffic. Thus flow F sends more than $M/100$ bytes. Since we are randomly sampling each byte with probability $10,000/M$, the probability that F will not be in the flow memory at the end of S seconds is $(1-10000/M)^{M/100}$ which is very close to e^{-100}. Notice that the factor of 100 in the exponent is the oversampling factor. Better still, the probability that flow F is in the flow memory after sending 5% of its traffic is, using a similar analysis, $1 - e^{-5}$ which is greater than 99% probability. Thus with 99% probability we will measure flow F's sent bytes as being only 5% less than its actual amount.

It should be clear that the analysis can be generalized to arbitrary threshold values; the memory needs scale inversely with the threshold percentage. Notice also that the analysis assumes that there is always space to place a sample flow not already in the memory. Setting $p = 10,000/M$ ensures that the average number of flows sampled is no more than 10,000 but does not guarantee

[1] Most router vendors already generate random numbers using say LFSRs because of algorithms like RED

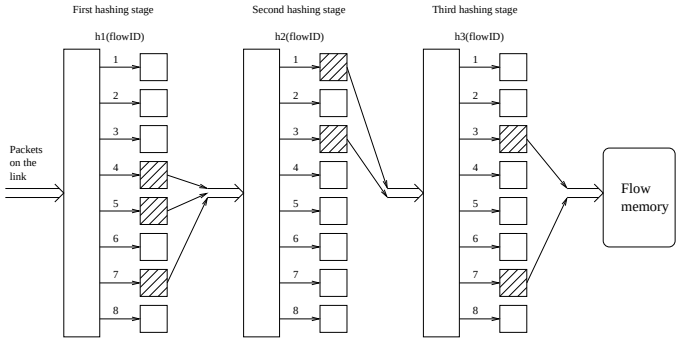

Fig. 2. Serial multistage filter: packets that hash to large buckets are passed to the next stage

that the total number does not exceed. Since the standard deviation in a binomial distribution is small, using a slightly larger memory size makes it extremely unlikely that the flow memory will overflow.

When compared to Cisco Sampled NetFlow our idea has four major differences. First, we sample randomly and not periodically. Second, we pick the sampling probability based on the specified threshold so that the memory will not overflow with high probability. Third, we take into account packet lengths by uniformly sampling bytes and sampling packets based on length. Fourth, we sample only to decide whether to add a flow to the memory; from that point on, we update the flow memory with every byte is sends.

We can provide counterexamples to show that each of our differences with NetFlow can cause NetFlow estimates to be inaccurate. Clearly, periodic sampling can be foiled by bursty traffic that arrives in a clump. Not taking into account packet sizes will cause the measurement algorithm to identify with higher probability flows using small packets than flows sending the same amount of traffic using large ones. Finally, simply estimating a flow's rate as the sum of its sampled bytes multiplied by the inverse of the sampling rate has a very high variance and cannot be used reliably for accounting. Note that in our scheme, by contrast, the estimated traffic is a lower bound on the customer's actual traffic sent.

B. Multistage filter counting

Our sampled counting can efficiently calculate the flows that take more than 1% of the link bandwidth but it does not seem to generalize to finding flows that take more than 1% of the total traffic. The following counting method can do so, and also totally avoid false negatives.

The basic multistage filter is illustrated in Figure 2. The building blocks are hash stages. Each stage computes an *independent* hash function on the flow ID of the incoming packets and decides based on it what bucket to add it to. Many flows will typically hash onto a bucket in each stage. Each bucket has a counter which counts the number of bytes sent in the current time period by all flows that hash onto that bucket. We define a threshold $T = M/100$. Flows that send more than this threshold should be added to the flow memory. All counters are reset to 0 at the start of each measurement period of S seconds. Buckets whose counter is above T/d (d is the number of stages) are considered large. Clearly, large flows are guaranteed to hash to large buckets at each of the stages.

The multistage serial filter passes on to the next stages only the flows that are in these large buckets. This filters out all the small flows that were not lucky enough to hash into a large bucket. If the number of buckets is large enough, the stage will discard a large percentage of the small flows. This filtering effect is amplified by applying multiple hash stages because the hash functions at different stages are independent. Thus in Figure 2, in the first stage, buckets 4, 5 and 7 are above threshold. We pass on to the second stage *only those packets that hash to buckets 4, 5, and 7.* In the second stage, buckets 1 and 3 are large. We pass on to stage three all the packets that hash to any of these buckets at stage two (and that passed stage one). The large buckets at stage three are 3 and 7. The packets that pass this stage too are delivered to the flow memory that keeps per flow state.

The parallel multistage filter differs slightly from the serial one presented above. The only difference is how it updates the buckets: the parallel filter updates the buckets at all stages when a packet comes in, irrespective of the packet being passed through or not. The packet is passed to the flow memory only if *all* the buckets it hashes to are above T. With the same number of stages and buckets as a serial filter, a parallel one, provides weaker filtering, but it is easier to analyze.

The following scenario shows the potential power of a multistage filter. Assume a 100 Mbytes/s link, with $100,000$ flows with a simple bimodal distribution of data rates. Up to 100 of them are large and the rest are small. The large flows use 1% of the link each, and the rest of the link is divided evenly between the small flows. We use stages of 1000 buckets and set the threshold to 1 Mbytes/s (the size of the large flows). The small flows (of size at most 1 Kbytes/sec) divide relatively evenly, with an average of 100 flows (at most 100 Kbytes/sec) per bucket. Assuming that it is very unlikely that a bucket gets filled with 10 times the average number of small flows, the only way for a small flow to pass stage 1 is to hash into the bucket with a large flow. Since the number of large flows is at most 100, we can bound the probability of a small

flow passing a stage by $100/1000 = 0.1$. With 4 independent stages, the likelihood that a small flow pass all 4 stages is at most 0.1^4. Thus the expected number of small flows that pass all 4 stages is at most $100,000 * 10^{-4} = 10$. With a small increase of the flow memory, this small number of small flows can be accounted for and discarded after observed for some time. More importantly, note the potential scalability of the scheme. To increase the number of flows to 1 million, we simply add a fifth hash stage to get the same effect. Thus to handle 100,000 flows, requires roughly 4100 memory locations, while to handle 1 million flows requires roughly 5100 memory locations, which is logarithmic scaling.

The basic intuition above can be generalized and captured in a number of theorems. We present the most important ones in appendix A. *We have theorems that make no assumptions about traffic distributions though use of Zipf distributions can sharpen the bounds slightly.* Note also that the analysis above of multistage filtering assumed the threshold was set based on 1% of the total bandwidth. We can use multistage filtering to find flows that take more than 1% of the total traffic sent during a measurement period. However, in this case we only know the threshold after the measurement period finishes. Thus we can identify flows that were large under the assumption that such flows send at least one packet in the next measurement period.

III. Possible Optimizations

Ideally, the flow memory requires a CAM to work at high speeds. The size of this CAM can be reduced by using a hash table which supports a bounded number of collisions and using a small CAM as a backup This is very similar to the use of fully associative victim caches to back up a large direct mapped or set associative cache in architecture.

So far we have assumed that traffic is measured in measurement periods that are arbitrary (1 second, 1 hour, or even 1 day) assuming sufficiently large counters. However, this has two problems. First, it requires initializing counters at the start of each period. Second, flows that send traffic across measurement periods may not be caught accurately. A solution is to handle the counters of the multistage filters as leaky buckets decrementing them gradually.

Extensions to the multistage filter which we omit for brevity can allow us to estimate with good precision the number of all active flows and the standard deviation of flow sizes.

IV. Applications for Large Flow Identification

Once we can identify large flows using a small amount of state, we can apply such a technique to several applications besides deriving traffic patterns. Some new applications we can envisage include:

- *Scalable Threshold Accounting:* The two poles of pricing for network traffic are usage based or duration based. While usage-based pricing [10], [11] has been shown to improve overall utility, usage based pricing is not scalable because of the large number of flows. Consider, instead, a scheme where we measure all traffic that is more than $x\%$ of the link [2]; such traffic is subject to usage based pricing, the remaining traffic is subject to duration based pricing. By changing the value of x from 0 to 100, we can move from pure usage based pricing to pure duration based pricing. More importantly, we believe that for reasonably small values of x (say 0.1%) our algorithm can offer a compromise between the two extremes that is scalable and yet offers almost the same degree of utility as usage based pricing.
- *Real-time Traffic Monitoring:* Many ISP backbones monitor traffic to look for hot-spots that can rerouted using MPLS tunnels or paths through reconfigurable optical switches. Similarly, other ISPs may look for sudden increases into certain traffic types (e.g., TCP Resets, ICMP messages) sent to or from individual endnodes. These might indicate a denial of service attack.
- *Scalable Queue Management:* As we move further down the time scale, there are other applications that would benefit from identifying large flows. Scheduling mechanisms attempting to approximate (weighted) max-min fairness need to detect and penalize flows sending above their fair rate. Keeping per flow state only for such flows does not affect the fairness of the scheduling and can account for substantial state savings. This problem is actually more complicated because the definition of a non-conformant flow can depend on round-trip delays as well. Several papers address this issue including [15], [14], [12].

V. Related work

NetFlow is intended (by Cisco) to serve as a basis for usage based billing. Cisco's solution to the problem of NetFlow generating too much data (introduced in IOS 12.0(3)T) is to aggregate raw data using aggregation caches; only the aggregate data is exported. Sampled NetFlow [2] is recommended for routers with speeds over OC-

[2] There are a few lines referring to such an idea in [11], but the idea is not developed further, and there are no solutions to the technical problems it raises.

3 rates, where the performance penalty of updating the flow cache from DRAM is reduced by sampling every x packets, where x is a parameter.

The papers [3], [18] deal with correlating measurements taken at various points to find spatial traffic distributions; they are orthogonal to the techniques described in this paper. [3] mentions the idea of doing random sampling but this reduces the amount of processing, not the required memory to keep track of flow state, and loses information about flow rates. The papers in [13] and [14] use similar but different techniques to our parallel hash idea to compute different metrics (set intersections and drop probabilities).

There is related work in the database community. In [4], Gibbons and Matias define and analyze counting samples similar to the ones we propose. However, we compute a different metric, need to take into account packet lengths, and have to size memory in a different way. In [5], Fang et al look at efficient ways of exactly counting the number of appearances of popular items in a database. Among their mechanisms is the multi-stage filter that we propose. However, they use sampling as a front-end before the filter and use multiple passes. Thus their final algorithm and analysis is very different from ours.

REFERENCES

[1] Cisco NetFlow. http://www.cisco.com/warp/public/732/Tech/netflow/

[2] Sampled NetFlow. http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120s/120s11/12s_sanf.htm

[3] Deriving Traffic Demands for Operational IP Networks: Methodology and Experience, by Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Proceedings of ACM SIGCOMM, Stockholm, pp. 257-270, August 2000

[4] New sampling-based summary statistics for improving approximate query answers, by Philip B. Gibbons and Yossi Matias. Proceedings of ACM SIGMOD International conference on Management of Data, pp. 331-342, June 1998

[5] Computing Iceberg Queries Efficiently, by Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, Jeffrey D. Ullman. Proceedings of 24th International Conference on Very Large Data Bases, pp. 307-317, August 1998

[6] The space complexity of approximating the frequency moments, by Noga Alon, Yossi Matias and Mario Szegedy. Proceedings of 28th ACM Symposium on the Theory of Computing, pp. 20-29, May 1996

[7] Inter-AS Traffic Patterns and Their Implications, by Wenjia Fang and Larry Peterson. Proceedings of IEEE GLOBECOM, Rio, Brazil, December 1999

[8] Andrei Z. Broder and Anna R. Karlin: Multilevel Adaptive Hashing

[9] High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching, by T. V. Lakshman and D. Stiliadis, Proceeding of ACM SIGCOMM, September 1998

[10] Pricing the Internet, by J. Mackie-Masson and H. Varian, in B. Kain and J. Keller (eds.), "Public Access to the Internet", MIT Press, Cambridge, MA, 1995.

[11] Pricing in Computer Networks: Reshaping the Research Agenda, by S. Shenker, D. Clark, D. Estrin, S. Herzog, Telecommunications Policy, vol. 20, No. 3, April 96.

[12] Controlling High Bandwidth Flows at the Congested Router, by Ratul Mahajan and Sally Floyd, http://www.aciri.org/floyd/papers.html, Feb 2001.

[13] B. Bloom, "Space/time trade-offs in hash coding with allowable errors", Commun. ACM, vol. 13, no. 7, pp. 422-426, July 1970

[14] Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness, by Wu-chang Feng, Kang G. Shin, Dilip Kandlur, Debanjan Saha, IEEE Infocom, Anchorage, Alaska, April 22-26 2001

[15] The New Red Penalty Box, by Van Jacobson, Kathleen Nichols, and Li Fan, Proposal to End-to-end Research Group, July 2000.

[16] Random sampling for histogram construction: How much is enough? by S. Chaudhuri, R. Motwani and V. Narasayya, Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 436-447, June 1998

[17] Probabilistic counting, by P. Flajolet and G. N. Martin, Proceedings of 24th IEEE Symposium on Foundations of Computer Science, pp. 76-82, November 1983

[18] Trajectory Sampling for Direct Traffic Observation, by N.G. Duffield and M. Grossglauser. Proceedings of ACM SIGCOMM, Stockholm, pp. 271-282, August 2000

[19] K. Thomson, G. J. Miller, and R. Wilder. Wide-area traffic patterns and characteristics. *IEEE Network*, December 1997.

[20] Traffic Flow Measurement: Architecture, by N. Brownlee, C. Mills, and G, Ruth, RFC 2063, January 1997

[21] Packet Classification on Multiple Fields, P. Gupta and N. Mckeown, Proc SIGCOMM 99.

APPENDIX

I. FORMAL ANALYSIS OF PARALLEL MULTISTAGE FILTERS

We first define some notation.

- n the number of flows;
- n_{pass} the number of flows passing the filter;
- b the number of buckets;
- d the depth of the filter (the number of stages);
- T the threshold we use for deciding whether a bucket is large or not, expressed as number of bytes sent during the measurement interval;
- C the capacity of the link, expressed as number of bytes that can be sent during the measurement interval;
- k this is a factor that expresses the strength of the filtering. It is computed as the ratio of the threshold and the average size of a bucket: $k = \frac{T b}{C}$. In our example k was 10;
- s the size of a certain flow;
- p_s the probability for flow of size s to pass the filter;

Theorem 1: For any parallel multistage filter that uses as threshold for the buckets T the size of the smallest large

flow, all large flows will be detected.

Now that we know that we have no false negatives, we analyze the false positives. We will not go deeply into how one can distinguish the small flows that passed the filter from the large ones. What we are concerned with is the number of small flows that can pass the filter. This has implications on how we dimension the flow memory, because, even if we can tell after some time whether a flow passing the filter is large or small, the memory has to hold the small ones also until they are deemed to be of no interest.

Theorem 2: The expected number of flows passing a parallel multistage filter is bound by

$$E[n_{pass}] \leq max\left(\frac{b}{k-1}, n\left(\frac{n}{kn-b}\right)^d\right) + n\left(\frac{n}{kn-b}\right)^d \quad (1)$$

For our example, this would result in a bound of 121.2 flows. Using 3 stages would have resulted in a bound of 200.6 and using 5 would give 112.1. If using the same number of buckets, we would change our target from flows above 1% of the link to flows above 0.3% (decreasing k from 10 to 3), the bound would be 2502.3 flows. Using 5 stages the bound would be 918.5 flows and for 6 stages 639.9. We can see how, after the first term dominates the max, there is not much gain in further strengthening the filtering by introducing more stages.

Theorem 2 gives a bound that holds for all possible distributions of flow sizes. It has been observed that various types of network traffic (e.g. accesses to web servers) have a Zipf-like distribution of flow sizes. Making the assumption that flow sizes have a Zipf distribution leads to an even stronger bound. We need two lemmas before proceeding to the theorem.

Theorem 3: If the flows sizes have a Zipf distribution, the expected number of flows passing a parallel multistage filter is bound by

$$E[n_{pass}] \leq i_0 + \frac{n}{k^d} + \frac{db}{k^{d+1}} + \frac{db(i_0 - 0.5)^{d-1}}{ln(n+1)k^{d+1}} \quad (2)$$

where $i_0 = \lceil max(1.5 + \frac{b}{k\ ln(n+1)}, \frac{b}{ln(2n+1)(k-1)})\rceil$.

With this new bound, for our filter we obtain a new upper bound on the number of flows expected to pass of 25.1 ($i_0 = 11$).

A. High probability bounds on filtering

Above we gave bounds for the number of flows expected to pass the filter. Since we generously overestimate the probability of flows passing the filter, these numbers are more conservative than what we will see in practice. However, we want to give a stronger theoretical result that bounds the probability of the unlikely event that the number of passing flows considerably exceeds the expected number.

Theorem 4: With probability p_{safe} the number of flows passing the parallel cascade is bound by

$$n_{pass} \leq b - 1 + \lfloor n\left(\frac{1}{k-1}\right)^d - \frac{ln(p_{safe})}{3} + \sqrt{\frac{ln(p_{safe})^2}{9} - 2n\left(\frac{1}{k-1}\right)^d ln(p_{safe})}\rfloor$$

For our example, if we are willing to make a mistake in 10^6 measurement intervals, we can bound the number of flows passing by 1221. If we accept a mistake in 10^{20} measurement intervals, the number is 1286. These numbers are pretty far from the expected value. We can further refine the analysis to make the bound tighter. Unfortunately the result is not a closed form expression, but an algorithm we can use to numerically compute the bound. We will use this algorithm to compute bounds in both the distribution free case and in the case when we know we have a Zipf distribution of flow sizes. We omit for brevity the exact algorithm we used to compute the bound. We obtained that with probability at most 10^{-6} the number of flows passing the parallel cascade from the example is not more than 211 in the general case and 95 for a Zipf distribution. With probability at most 10^{-20} the number of flows passing is not more than 314 in the general case and 152 for a Zipf distribution.