Counting Flows over Sliding Windows in High Speed Networks

Josep Sanjuàs-Cuxart, Pere Barlet-Ros, and Josep Solé-Pareta

Universitat Politècnica de Catalunya (UPC), Computer Architecture Dept.
Jordi Girona, 1-3 (Campus Nord D6), Barcelona 08034, Spain
{jsanjuas,pbarlet,pareta}@ac.upc.edu

Abstract. Counting the number of flows present in network traffic is not trivial, given that the naive approach of using a hash table to track the active flows is too slow for the current backbone network speeds. Several algorithms have been proposed in the recent literature that can calculate an approximate count using small amount of memory and few memory accesses per packet. Fewer works have addressed the more complex problem of counting flows over sliding windows, where the main challenge is to continuously expire old information. One of the existing proposals is a straightforward adaptation of the direct bitmaps technique to the sliding window model. We present an algorithm called Countdown Vector that also builds upon the direct bitmaps technique. Our algorithm, however, obtains significant cost reductions both in terms of memory and CPU, by introducing an extra approximation in the mechanism in charge of the expiration of old information.

Keywords: traffic measurement, counting active flows, sliding windows.

1 Introduction and Related Work

The design of efficient algorithms to count the number of active flows in high-speed networks has recently attracted the interest of the network measurement community [1,2,3]. Counting the number of flows in real-time is particularly relevant to network operators and administrators for network management and security tasks. For example, this metric is the basis of most network intrusion detection systems to detect port scans and DoS attacks.

However, the naive solution of tracking flows using a hash table is becoming unfeasible in high-speed links. First, it requires several memory accesses per packet with the overhead of creating new flow entries and handling collisions. Second, this solution uses large amounts of memory, since it requires storing all flow identifiers. The number of concurrent flows present in high speed networks is very large, and can be well over one million in current backbone links [4]. Therefore, hash tables must be stored in DRAM, which has an access time greater than current packet interarrival times. For example, access times of standard DRAM are in the order of tens of nanoseconds, while packet interarrival times can be up to 32 ns and 8 ns in OC-192 (10 Gb/s) and OC-768 (40 Gb/s) links

L. Fratta et al. (Eds.): NETWORKING 2009, LNCS 5550, pp. 79–91, 2009.

respectively. Thus, flow counting algorithms must be able to process each packet in very few nanoseconds to be suitable for high-speed links.

In order to reduce the large amount of memory required to store flow tables, most routers (e.g., Cisco NetFlow [5]) and network monitoring systems [6] resort to packet sampling. However, it has been shown that packet sampling is biased towards large flows and tends to underestimate the total number of flows [7].

Recently, several probabilistic algorithms have been proposed to efficiently estimate the number of flows in high-speed networks [1,2,3]. These algorithms share the common approach of using specialized data structures to approximately count the number of flows, which need a very small amount of memory, as compared to the traditional approach of keeping per-flow state, while requiring one or very few memory accesses per packet. This drastic reduction in the memory requirements allows the storage of these data structures in fast SRAM, with access times below 10 ns. These techniques can therefore be implemented in router line cards or, in general-purpose systems, the data structures can reside in cache memory.

The direct bitmaps technique is the basis of most probabilistic algorithms to estimate the number of flows. This technique was first proposed by Whang et al. [8] in the database community and popularized in the networking community by Estan et al. in [1], which also presents several variants of the direct bitmaps that require less memory. One of the variants, called multiresolution bitmaps, obtains similar accuracy as Loglog counting [9]. Giroire's proposal [10] is to estimate the count from the minimum of the hashed values. These and other techniques are compared in [11]. A remarkable conclusion from this study is that, from a practical standpoint, direct bitmaps offer the best tradeoff between complexity and accuracy.

The basic idea behind direct bitmaps is to use a small vector of bits (i.e., bitmap). For each packet, a hash of the flow identifier is computed and the corresponding bit is set in the bitmap. At the end of a measurement interval, the number of flows can be simply estimated according to the number of non-set bits and the collision probability [8]. The main problem of these algorithms is that they can only operate over fixed, non-overlapping measurement intervals and, therefore, cannot obtain continuous estimates of the number of flows.

The *sliding window model* is increasingly gaining interest in the networking and database communities, given the streaming nature of many current data sources (e.g., network traffic, sensor networks or financial data). For example, a new class of systems is emerging in the database [12,13] and network monitoring [14,15,16] communities in order to support continuous queries over sliding windows. Under this new paradigm, queries process streaming data, instead of static databases, and compute metrics over time windows that advance continuously (e.g., the number of active flows during the last 5 minutes). An interesting introduction to the field of data stream research can be found in [17], while a more informal discussion that motivates this research area is [18].

Datar et al. [19] analyze the basic problem of counting the number of ones within the last N elements of an arbitrary stream composed of zeros and ones.

They present an algorithm based on a technique called Exponential Histograms that can provide an approximate solution to this problem. Using this basic algorithm as a building block, they approach related problems such as calculating sums, averages, maximum and minimum values. For the distinct count problem, they propose adapting a bitmap-based counting technique to the sliding window model by storing timestamps instead of bits in each bitmap position.

Kim and O'Hallaron [3] propose a technique that addresses the flow counting problem using this approach, called Timestamp Vector (TSV). TSV is based on the original direct bitmap algorithm and consists of replacing the bitmap for a vector of timestamps. However, the main limitations of TSV are that (*i*) it requires a significantly larger amount of memory compared to the original direct bitmap (a 64-bit timestamp for each vector position) and (*ii*) the cost of the queries is linear with the size of the vector, which renders this solution impractical in scenarios where a continuous estimate of the number of flows is needed.

In this paper, we propose a new algorithm called Countdown Vector (CDV) to estimate the number of flows over sliding windows. The basic idea behind our method is the use of a vector of small timeout counters, instead of full timestamps, that are decremented independently of the per-packet update and query processes. Our algorithm requires less resources (i.e., CPU and memory) than existing solutions, and has $O(1)$ query cost. This way, a network monitoring system can implement our method using less memory, and can react faster to changes in the number of flows (e.g., network anomalies or attacks), since queries can be issued more frequently than in previous proposals. Another interesting advantage of our technique over other alternatives is that it is possible to degrade the accuracy of the estimates according to a given CPU and memory budget.

We implement our algorithm in an existing network monitoring system and present experimental evidence of the accuracy and cost of our technique using real-world packet traces. Our results show that our technique is able to estimate the number of flows over a wide range of sliding windows with a similar accuracy to the TSV, while reducing the memory requirements to up to 1/20th and the number of memory accesses to up to 1/30th.

The rest of the paper is organized as follows. Section 2 describes our algorithm to estimate the number of flows over sliding windows, while Section 3 presents a performance evaluation of our technique using real packet traces collected at the access link of a large university network. Finally, Section 4 concludes the paper and discusses future work.

2 Countdown Vector Algorithm

This section describes our method to count flows over sliding windows. A flow is defined as a sequence of packets that share equal values for a subset of the TCP/IP header fields. Typically, a flow is identified by the 5-tuple that consists of the source and destination IP addresses and ports, and protocol field. However, our method is agnostic to the particular definition of a flow.

Since our algorithm is based on direct bitmaps, we review this technique in greater detail. We also describe the Timestamp Vector algorithm (TSV), which

Table 1. Notation

b: Number of positions of the vector or bitmap.
c: Value to which counters are initialized.
f: Time between queries in a jumping window model.
n: Number of flows in the traffic during a time window.
$\hat{n}$: Estimation of the number of flows n.
s: Time between counter updates.
w: Length of the time window.
z: Number of positions with value 0 in the bitmap or the vector or bitmap.

we use to compare the accuracy and overhead of our method. Additionally, we propose a simple improvement of the TSV that significantly reduces its memory requirements under certain conditions.

2.1 Direct Bitmaps

Like the naive algorithm of using a hash table to track flows, direct bitmaps are based on hashing, but do not create one table entry per flow. Instead, they just record whether a position in the hash table would be used or not (hence the name bitmap). The algorithm uses a pseudo-random hash function (e.g., [20]) to evenly distribute the positions corresponding to each flow identifier.

One could think of extrapolating the number of ones in the bitmap as the number of flows in the original dataset. However, this would not be correct, since there is a non-negligible probability that several flow identifiers collide (i.e., hash to the same bitmap position), given the reduced size of the bitmap. While the bitmap cannot be used to extract the exact count of flows, instead, an estimate can be obtained from the bitmap size (b) and the number of zeros in the bitmap (z) as shown in [8]:

$$\hat{n} = b \; ln\left(\frac{b}{z}\right) \tag{1}$$

We refer to the process of obtaining an estimate of the number of flows as the process of *querying* the bitmap. Table 2.1 summarizes the notation used throughout this section.

The principal advantage of this technique is that, with a small amount of memory (especially when compared to the naive algorithm), the number of flows can be estimated with high accuracy. A second important characteristic of this approach is that the accuracy can be arbitrarily increased or reduced by the bitmap size. To correctly dimension the bitmap, the appropriate size must be chosen so that the expected amount of unique elements can be estimated within the desired error bounds, as explained in [8].

To give the reader an idea of how little memory this algorithm requires, it suffices to state that this technique can count 10^6 elements by using around 20KB with errors below 1% [8].

2.2 Timestamp Vector

As discussed in the previous section, direct bitmaps are a very efficient technique to count the amount of flows in the network traffic. However, in order to provide meaningful values when monitoring a network traffic stream, measurement statistics must be bounded in time, i.e., must correspond to a particular *time window*. Direct bitmaps do not incorporate a sense of time and thus must be periodically reset to avoid lifetime flow counting.

Periodically querying and resetting a direct bitmap would provide flow counts over consecutive, non-overlapping windows. While there is value to this application of the technique, it imposes the restriction that queries must be aligned with bitmap resets. In contrast, in the *sliding window* model, queries can arrive at any time. This requirement implies that old information must be removed as newer arrives, in order to continuously maintain the data structures to be able to provide an estimate at any time.

A straightforward solution to adapt the direct bitmaps to the sliding window model is the Timestamp Vector algorithm [3]. Instead of a bitmap, a vector of timestamps is now used. When a packet hashes to a particular position, its timestamp is set to the timestamp of that packet. Using this vector, when a query for a time window of w time units arrives at time t, the number of flows can be estimated by using Equation 1, where in this case z corresponds to the number of positions with timestamp less than $t - w$, and b to the number of positions in the vector. Note that this requires a full traversal of the vector for each query.

2.3 Extension of the Timestamp Vector

The principal limitation of the Timestamp Vector technique is the increase in the amount of memory that it requires. Timestamps in network monitoring are typically 64 bits long, e.g. as provided by libpcap [21] or Endace DAG cards [22]. Hence, the final size of the vector is increased by a factor of 64 compared to the original bitmap.

A relaxation of the sliding window model is the *jumping window* model [23], where the window does not advance continuously, but discretely in fractions of the measurement window. When operating under this model, we propose the following improvement over the Timestamp Vector algorithm that significantly reduces its memory requirements. Instead of full timestamps, only the fraction of the window where the packet arrived is stored. This idea can be implemented using $log_2\left(w/f + 1\right)$ bits per position when measuring a window of w time units with query periods of f. For example, with a window of 30 s and queries every 1 s, the original Timestamp Vector will require 64 bits per position, while this approach would require only 5 bits per position, using below 10% of the memory. Note however that, using this extension, the Timestamp Vector can only be queried every f time units.

One limitation of both variants of the Timestamp Vector algorithm is, thus, their additional memory requirements. The jumping window variant reduces memory

usage by limiting the time where queries may be performed. The second disadvantage of this scheme is that, for each query, one full traversal of the array is required to calculate the number of positions whose timestamp is older than $t - w$.

2.4 Countdown Vector

In this section we present our technique for flow counting over sliding windows. Our scheme is, like the TSV algorithm, an adaptation of the direct bitmaps to the sliding window model. We start by outlining the basic intuition behind the technique we propose.

The main difficulty when adapting the direct bitmap to the sliding window model is to remove old information from the data structure as time advances. Let us start by defining an ideal algorithm which would precisely calculate z in a sliding window of w time units. Recall that b (vector size) and z (number of zeros in it) suffice to estimate the number of flows in the traffic using Equation 1. A vector of b positions could be used, with the values set to w time units every time a packet hashed to the corresponding position. Every time unit, all the positions of the vector with non-zero value would be decremented by one. The count of positions with counter value zero would correspond to z in order to estimate the number of flows seen in the time window.

In order to obtain a perfect resolution, this scheme would require defining the time unit to the maximum resolution of the system clock. This ideal scheme would therefore be very costly in terms of both memory and CPU. First, a high resolution counter would have be stored in each vector position, thus increasing the overall memory required to store the vector. Second, all of the counters would need to be updated for each time unit. These additional costs make this technique infeasible as described, especially when it would achieve results equivalent to the TSV.

The technique that we propose is very similar to the ideal algorithm just described but, instead, using small integer counters in each position. Using small values requires less memory and calls for a low counter decrement frequency for counters to reach zero after w time units, in exchange of introducing small inaccuracies. In the following paragraphs we describe our technique with detail. The key to the effectiveness of our algorithm is that surprisingly low values suffice to achieve good accuracy to estimate network flow counts. In practice, then, we require less memory and introduce lower overhead compared to the original TSV algorithm and to its extension presented in Section 2.3.

Algorithm. Our algorithm starts by allocating a vector of counters, all of which are initialized to zero. Our algorithm can then be seen as divided in two concurrent processes: the first updates the vector for each packet, while the second is in charge of decreasing the counters at a fixed rate.

The first process is described in Algorithm 1 and runs synchronously with the packet stream. When a packet hashes to a position, we store a maximum value c in the corresponding position. This process remains the same independently of the time window that is being measured.

In contrast, the second process, which is described in Algorithm 2, performs a continuous maintenance of the vector. It decreases one counter every s time units,

Algorithm 1. Initialization and packet-synchronous operations

```
   Data: b: vector size, c: counter initialization value
 1 z ← b;
 2 vector ← {0, ..., 0};
 3 start continuous maintenance procedure;
 4 foreach packet p do
 5     key ← hash(p.flow_identifier);
 6     if vector[key mod b] = 0 then          /* update count of zeros */
 7         z ← z − 1;
 8     end
 9     vector[key mod b] ← c;
10 end
```

advancing one position in the vector at every step. The desired time window w plays a role in this data structure maintenance process, where it conditions the speed at which counters are decreased.

To determine s we proceed as follows. Since the packet arrivals and the maintenance are independent processes, and each flow hashes to a random position, on average, the first decrement of a counter (after it is set to w by the first process) will happen after $b/2$ counter updates. Afterwards, the counter will be decremented after b additional counter updates. Therefore, on average, counters reach zero after $b/2 + b\ (c-1) = b\ (c - 1/2)$ updates. Since this time must correspond to the time window w, we calculate s the following way:

$$s = \frac{w}{b\ \left(c - \frac{1}{2}\right)} \tag{2}$$

Both the first and the second processes maintain the count of positions of the vector with value zero, updating the value of z as values of the vector are modified. This has the advantage that query operations run in constant time $O(1)$, by simply applying Equation 1, as described in Algorithm 3.

Our algorithm is then governed by the following configuration parameters: (*i*) the desired measurement window w, (*ii*) the size of the vector b, and (*iii*) the maximum values to which counters are set c.

The precision of our algorithm increases with larger b and c values. Larger values of b (vector size) make the estimation error of Equation 1 decrease, as explained in further detail in [8]. On the other hand, increasing c also has a positive impact on the accuracy of the method, since, the larger c is, the more our algorithm approaches the ideal algorithm explained in the previous subsection.

However, larger values of c increase both the memory and CPU requirements of our algorithm, since, the higher c is, the more space is required to store the counters, and the higher the counter decrease frequency, i.e., s decreases, according to Equation 2.

Our algorithm has two sources of error. First, the approximation introduced by the original technique upon which ours builds, the direct bitmaps. The second source of error is introduced by the fact that old information is expired after w

Algorithm 2. Continuous maintenance procedure

```
   Data: b: vector size, c: counter initialization value, vector: vector of counters,
   z: count of positions with value 0
 1 s ← w / (b×(c−1/2));
 2 i ← 0;
 3 while True do
 4     sleep for s time units;
 5     if vector[i] = 1 then                    /* update count of zeros */
 6         z ← z + 1;
 7     end
 8     vector[i] ← max(0, vector[i] − 1);
 9     i ← (i + 1) mod b;
10 end
```

Algorithm 3. Query procedure

```
   Data: b: vector size, z: number of zeros in vector
 1 return b × ln(b/z);
```

time units only *on average*. Inevitably, some counters will, on the worst case, be set to c right after the maintenance process has decremented the corresponding position, and will thus will reach zero after $c * b * s = \frac{c}{c-1/2}w$ time units. Conversely, others will reach zero after $\frac{c-1}{c-1/2}w$ time units. In the worst case a counter will be inaccurate only during this small period of time. This explains why the accuracy increases with larger values of c, which tighten these bounds around w.

In order to choose a value for b, the tables provided by [8] can be looked up to determine the an appropriate vector size for the expected number of flows in the traffic. In the next section we analyze the impact of c over the accuracy of the method and show the overhead reduction of our method compared to both variants of the Timestamp Vector.

3 Evaluation

In order to obtain sensible results, we have tested our technique using real traffic. We collected a 30-minute packet-level traces in November 2007 at the access link of the Technical University of Catalonia (UPC), which connects 10 campuses, 25 faculties and 40 departments to the Internet through the Spanish Research and Education network (RedIRIS). The trace accounts for 106M packets with an average data rate of 271.6 Mbps. The average number of flows is around 50000 in a ten seconds window, and 1.8 million in a 10 minute window. Our technical report version of this paper includes the results obtained with other traces [24], which are similar to the ones presented in this section.

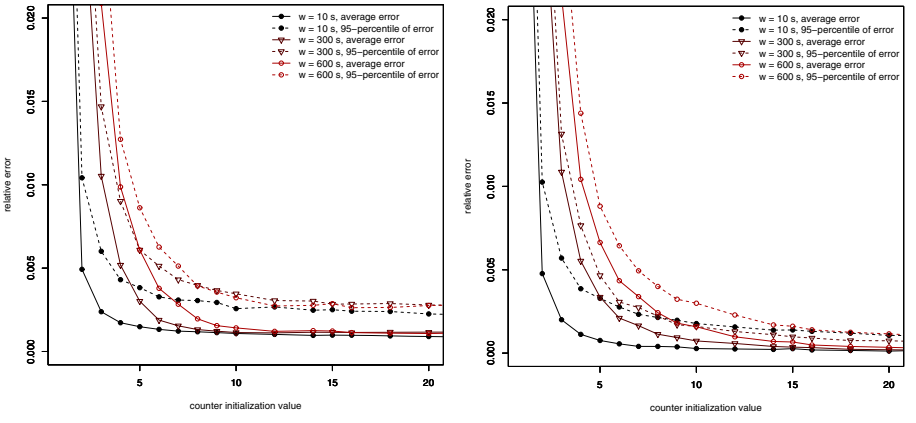

Fig. 1. Error of the Countdown Vector algorithm (left) and error of the Countdown Vector algorithm compared to the estimates of the Timestamp Vector algorithm (right)

Figure 1 (left) shows the results of running our algorithm with window sizes of 10, 300 and 600 seconds with different counter initialization values, using a fixed vector size. The size of the vector has been chosen according to [8] so that the average number of flows for the largest window can be counted with errors below 1%. Each point in the figure corresponds to a full pass on aforementioned trace, querying the algorithm every second, and shows either the average relative error or the 95th percentile of the relative error compared to a precise calculation of the number of flows.

As expected, for every window size, the relative error decreases as c increases. It is interesting to observe that, while the error is intolerable for very small values of c (5 and below), when it reaches values as small as 10 the error stabilizes, and does not decrease significantly beyond that point, even for a window as large as 600 s. This observation explains the great overhead savings that our technique shows in comparison to the Timestamp Vector, as we show in the next paragraphs. We have observed that the error for these values of c is almost equal to that of the Timestamp Vector algorithm. This source of error can be imputed to the underlying method of estimation of the direct bitmaps (see Equation 1). Figure 1 (right) shows the error of our method relative to the values obtained using the Timestamp Vector algorithm and confirms this observation.

We now examine the cost of the Countdown Vector algorithm and compare it to that of the Timestamp Vector. Figures 2 and 3 summarize a different set of executions of the algorithms using the same trace. To obtain realistic overhead calculations, in this case we dimension the vectors for both TSV and CDV appropriate for the observed number of flows in each time window, using [8], bounding the error introduced by the estimation formula to 1%. We dimension our variant of the TSV for a 1 second query frequency. For performace reasons, our implementation restricts the vector sizes to powers of two; we choose the smallest suitable sizes.

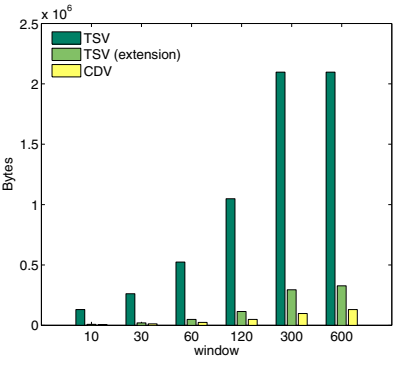

Fig. 2. Memory consumption for the three algorithms with varying window sizes

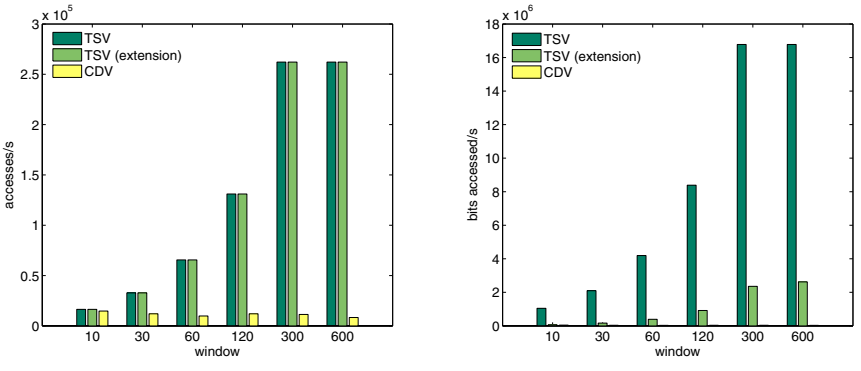

Fig. 3. Number of memory accesses per second (left) and number of bits accessed per second (right) for the three algorithms with varying window sizes

Our algorithm has, besides the size of the vector, an additional parameter: the counter initialization value (c). We have run the experiments with various c values, and have chosen, for each time window, the smallest that obtains at most 0.1% more relative error than the Timestamp Vector algorithm.

Figure 2 shows the high memory savings that our variant of the TSV introduces, at the expense of reducing the query frequency to only one second. In contrast, the CDV we propose further reduces the memory to roughly one half compared to our TSV variant, without introducing such a restriction.

We compare the cost of the algorithms using the number of memory accesses per second, assuming one query every second, and including the maintenance cost in the case of the CDV. However, we omit the cost of updating the vectors for every packet, since this cost is common to all of the algorithms. It suffices to state that it is only in the order of 50000 accesses per second, which roughly corresponds to the average packet rate in our trace. In Figure 3 (left), it can be observed that the cost of the Timestamp Vector grows with the size of the

vector, since it has to be traversed for every query. In contrast, the cost of the Countdown Vector remains small.

This figure is unfair to our variant of the TSV since, while the number of memory accesses are equal to those of the original TSV, these are accesses to smaller chunks of memory. Depending on the architecture, then, specific optimizations could be employed to improve its performance (e.g., read several positions with a single memory access). To correct this, we also present the cost in terms of bits accessed per second in Figure 3 (right).

The cost of both variants of the TSV grows proportionally to the vector size, since full vector traversals per query are required. Vectors grow with window sizes, since higher flow counts have to be obtained. In contrast, the CDV's query cost is constant; in our algorithm, the bulk of the cost is in the maintenance phase. However, since very low counter values can obtain an accuracy very similar to the TSV variants, counter decrements are performed at a low frequency, therefore incurring significantly lower costs.

4 Summary and Future Work

We have presented an algorithm called Countdown Vector that efficiently calculates the number of flows present in the network traffic over sliding windows. Our scheme introduces a continuous maintenance cost but, unlike previous proposals, can be queried in constant time. We have performed an evaluation using real traffic, and compared the cost in terms of memory and CPU to the state of the art Timestamp Vector algorithm. The Countdown Vector shows comparable accuracy with significatly lower costs.

The basic idea behind our scheme is to use a vector of timeout counters that expires old information in an approximate fashion. While in this work we implement this idea on the direct bitmaps technique, we plan on adapting our scheme to other more efficient flow counting algorithms. In particular, this scheme can be easily applied to the algorithms proposed by Estan et al. in [1].

Another important piece of future work is to perform a theoretical analysis of the accuracy of the algorithm we propose. While we have performed an empirical analysis using network traffic traces, it is important to be able to provide error bounds that apply to other scenarios.

Acknowledgments

This work was partially funded by the Spanish Ministry of Science and Innovation (MICINN) under contract TSI2005-07520-C03-02 (CEPOS). The authors thank the Supercomputing Center of Catalonia and UPCnet for allowing them to collect the packet traces used in this work. Authors are also grateful to Manel Martinez Torres and Gianluca Iannaccone for useful discussion in early stages of this work.

References

1. Estan, C., Varghese, G., Fisk, M.: Bitmap algorithms for counting active flows on high speed links. In: Proc. of ACM SIGCOMM Internet Measurement Conf. (October 2003)
2. Fusy, E., Giroire, F.: Estimating the number of Active Flows in a Data Stream over a Sliding Window. In: Proc. of SIAM Workshop on Analytic Algorithmics and Combinatorics (January 2007)
3. Kim, H., O'Hallaron, D.: Counting network flows in real time. In: Proc. of IEEE GLOBECOM (December 2003)
4. Fang, W., Peterson, L.: Inter-AS traffic patterns and their implications. In: Proc. of IEEE GLOBECOM (December 1999)
5. Cisco Systems: NetFlow services and applications. White Paper (2000)
6. Barlet-Ros, P., Iannaccone, G., Sanjuàs-Cuxart, J., Amores-López, D., Solé-Pareta, J.: Load shedding in network monitoring applications. In: Proc. of USENIX Annual Technical Conf. (June 2007)
7. Duffield, N., Lund, C., Thorup, M.: Properties and prediction of flow statistics from sampled packet streams. In: Proc. of ACM SIGCOMM Internet Measurement Workshop (November 2002)
8. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Trans. Database Syst. 15(2) (June 1990)
9. Durand, M., Flajolet, P.: Loglog Counting of Large Cardinalities. In: Proc. of Annual European Symposium on Algorithms (September 2003)
10. Giroire, F.: Order statistics and estimating cardinalities of massive data sets. In: Proc. of Intl. Conf. on Analysis of Algorithms (June 2005)
11. Metwally, A., Agrawal, D., El Abbadi, A.: Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In: Proc. of Intl. Conf. on Extending Database Technology: Advances in Database Technology (March 2008)
12. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (June 2002)
13. Golab, L., Özsu, T.M.: Issues in data stream management. SIGMOD Record 32 (June 2003)
14. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: A stream database for network applications. In: Proc. of ACM SIGMOD (June 2003)
15. Iannaccone, G.: Fast prototyping of network data mining applications. In: Proc. of Passive and Active Measurement Conf. (March 2006)
16. Reiss, F., Hellerstein, J.M.: Declarative network monitoring with an underprovisioned query processor. In: Proc. of IEEE Intl. Conf. on Data Engineering (April 2006)
17. Muthukrishnan, S.: Data Streams: Algorithms And Applications. Now Publishers Inc. (2005)
18. Hayes, B.: The Britney Spears Problem, `http://www.americanscientist.org/issues/pub/the-britney-spears-problem`
19. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. In: Proc. of ACM-SIAM Symp. on Discrete Algorithms (January 2002)
20. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. 18 (April 1979)

21. Jacobson, V., Leres, C., McCanne, S. (libpcap) Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release (June 1994), http://www.tcpdump.org
22. Endace: DAG network monitoring cards, http://www.endace.com
23. Golab, L., DeHaan, D., Demaine, E., Lopez-Ortiz, A., Munro, J.: Identifying frequent items in sliding windows over on-line packet streams. In: Proc. of ACM SIGCOMM Internet Measurement Conf. (October 2003)
24. Sanjuàs-Cuxart, J., Barlet-Ros, P., Solé-Pareta, J.: Counting network flows over sliding windows in high-speed networks. UPC Technical Report, http://loadshedding.ccaba.upc.edu/papers/counting-network-flows.techrep2008.pdf