Network data streaming: a computer scientist's journey in signal processing

Jun (Jim) Xu
Networking and Telecommunications Group
College of Computing
Georgia Institute of Technology

joint work with Abhishek Kumar, Min-Ho Sung, Qi Zhao, Zhen Liu (IBM), Mitsu Ogihara (Rochester), Haixun Wang (IBM), Jia Wang (AT&T), and Ellen Zegura

Outline

- Motivation and introduction
- Our representative data streaming works
  - Single-stream single-goal (like SISD)
  - Multiple-stream single-goal (like SIMD)
  - Multiple-stream multiple-goal (like MIMD)
- A fundamental hardware primitive for network data streaming
- A sampler of MSSG

Motivation for new network monitoring algorithms

Problem: we often need to monitor network links for quantities such as

- Elephant flows (traffic engineering, billing)
- Number of distinct flows, average flow size (queue management)
- Flow size distribution (anormaly detection)
- Per-flow traffic volume (anormaly detection)
- Entropy of the traffic (anormaly detection)
- Other “unlikely” applications: traffic matrix estimation, P2P routing, IP traceback

The challenge of high-speed network monitoring

- Network monitoring at high speed is challenging
  - packets arrive every 25ns on a 40 Gbps (OC-768) link
  - has to use SRAM for per-packet processing
  - per-flow state too large to fit into SRAM
- Traditional solution using sampling:
  - Sample a small percentage of packets
  - Process these packets using per-flow state stored in slow memory (DRAM)
  - Using some type of scaling to recover the original statistics, hence high inaccuracy with low sampling rate
  - Fighting a losing cause: higher link speed requires lower sampling rate

Network data streaming – a smarter solution

- **Computational model:** process a long stream of data (packets) in one pass using a small (yet fast) memory
- **Problem to solve:** need to answer some queries about the stream at the end or continuously
- **Trick:** try to remember the most important information about the stream *pertinent to the queries* – learn to forget unimportant things
- **Comparison with sampling**: streaming peruses every piece of data for most important information while sampling digests a small percentage of data and absorbs all information therein.

The “hello world” data streaming problem

- Given a long stream of data (say packets), count the number of distinct elements (F_0) in it
- Say in a, b, c, a, c, b, d, a – this number is 4
- Think about trillions of packets belonging to billions of flows ...
- A simple algorithm: choose a hash function h with range (0,1)
- $\hat{X} := min(h(d_1), h(d_2), ...)$
- We can prove $E[\hat{X}] = 1/(F_0 + 1)$ and then estimate F_0 using method of moments
- Then averaging hundres of estimations of F_0 up to get an accurate result

Data Streaming Algorithm for Estimating Flow Size Distribution [Sigmetrics04]

- **Problem:** To estimate the probability distribution of flow sizes. In other words, for each positive integer i, estimate n_i, the number of flows of size i.
- **Applications:** Traffic characterization and engineering, network billing/accounting, anomaly detection, etc.
- **Importance:** The mother of many other flow statistics such as average flow size (first moment) and flow entropy

- **Definition of a flow:** All packets with the same flow-label. The flow-label can be defined as any combination of fields from the IP header, e.g., $<$Source IP, source Port, Dest. IP, Dest. Port, Protocol$>$.

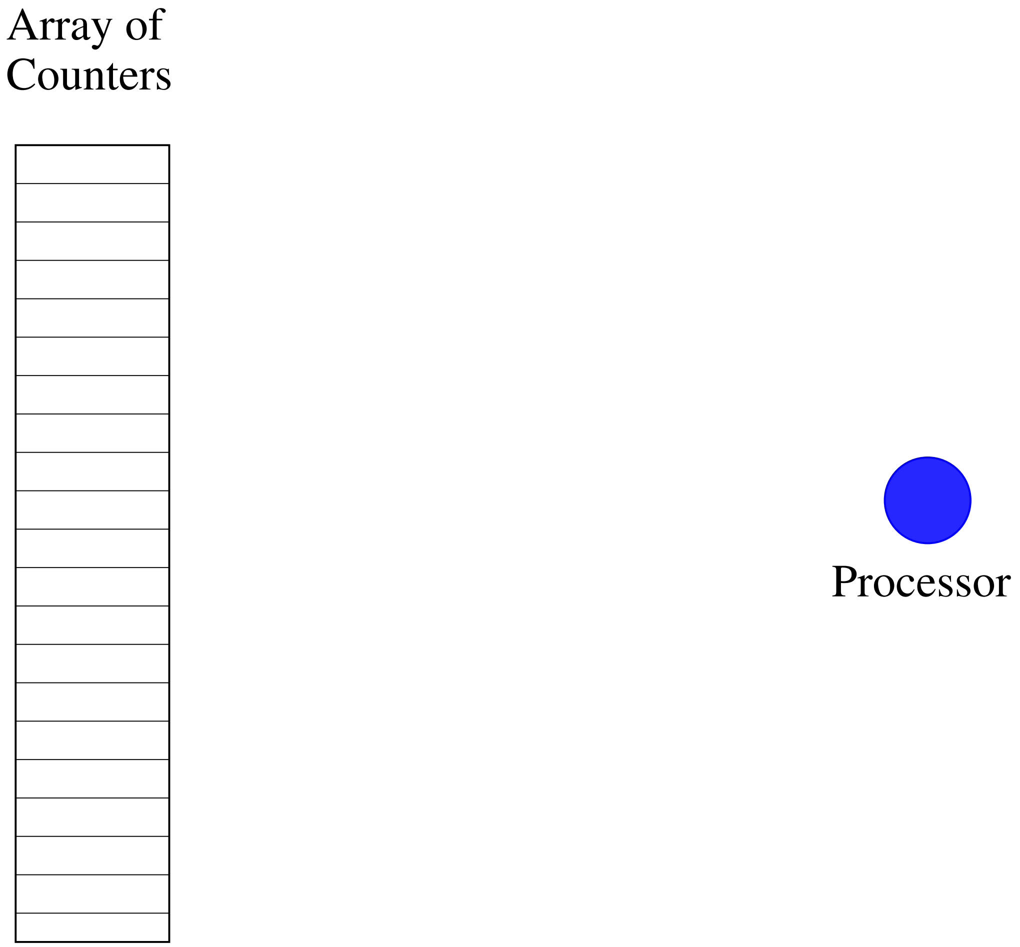
Our approach: network data streaming

- **Design philosophy:** "Lossy data structure + Bayesian statistics = Accurate streaming"
  - Information loss is unavoidable: (1) memory very small compared to the data stream (2) too little time to put data into the "right place"
  - Control the loss so that Bayesian statistical techniques such as Maximum Likelihood Estimation can still recover a decent amount of information.

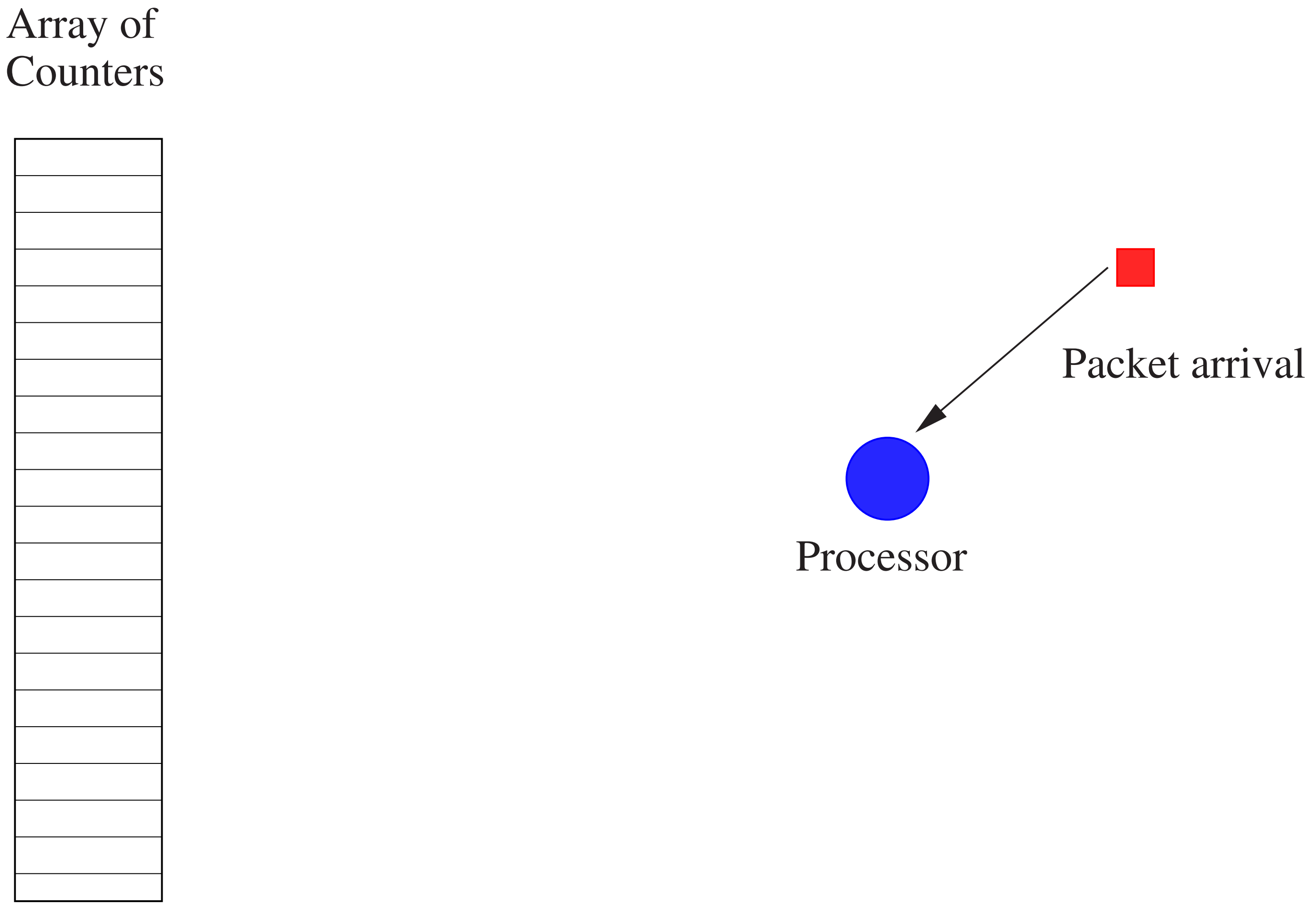
Architecture of our Solution — Lossy data structure

- Maintain an array of counters in fast memory (SRAM).
- For each packet, a counter is chosen via hashing, and incremented.
- No attempt to detect or resolve collisions.
- Each 64-bit counter only uses 4-bit of SRAM (due to [Zhao, Xu, and Liu 2006])
- Data collection is lossy (erroneous), but very fast.

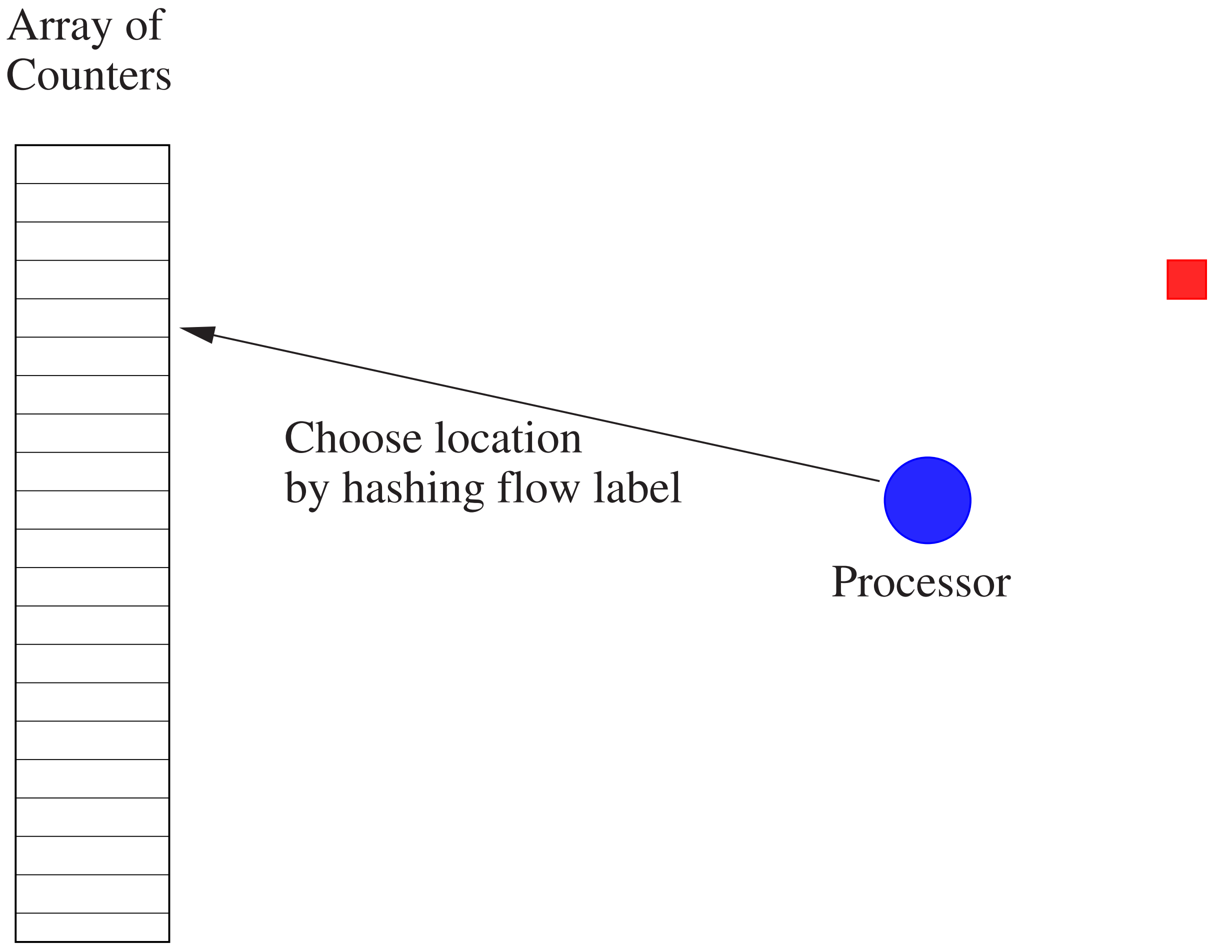
Counting Sketch: Array of counters



Counting Sketch: Array of counters

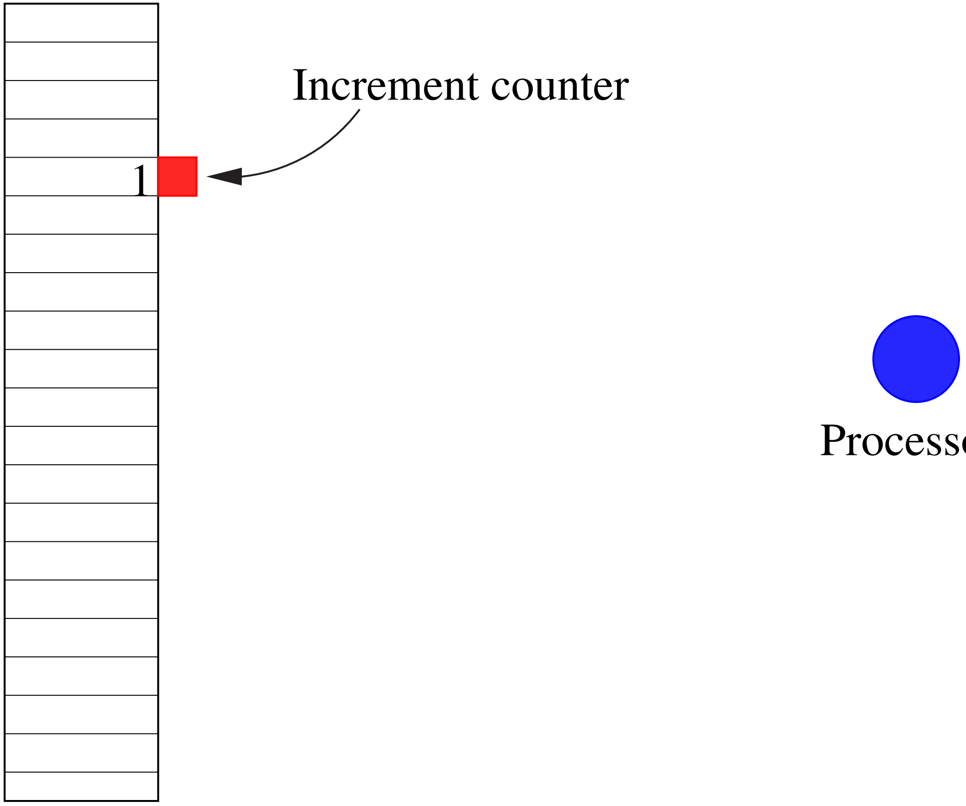


Counting Sketch: Array of counters

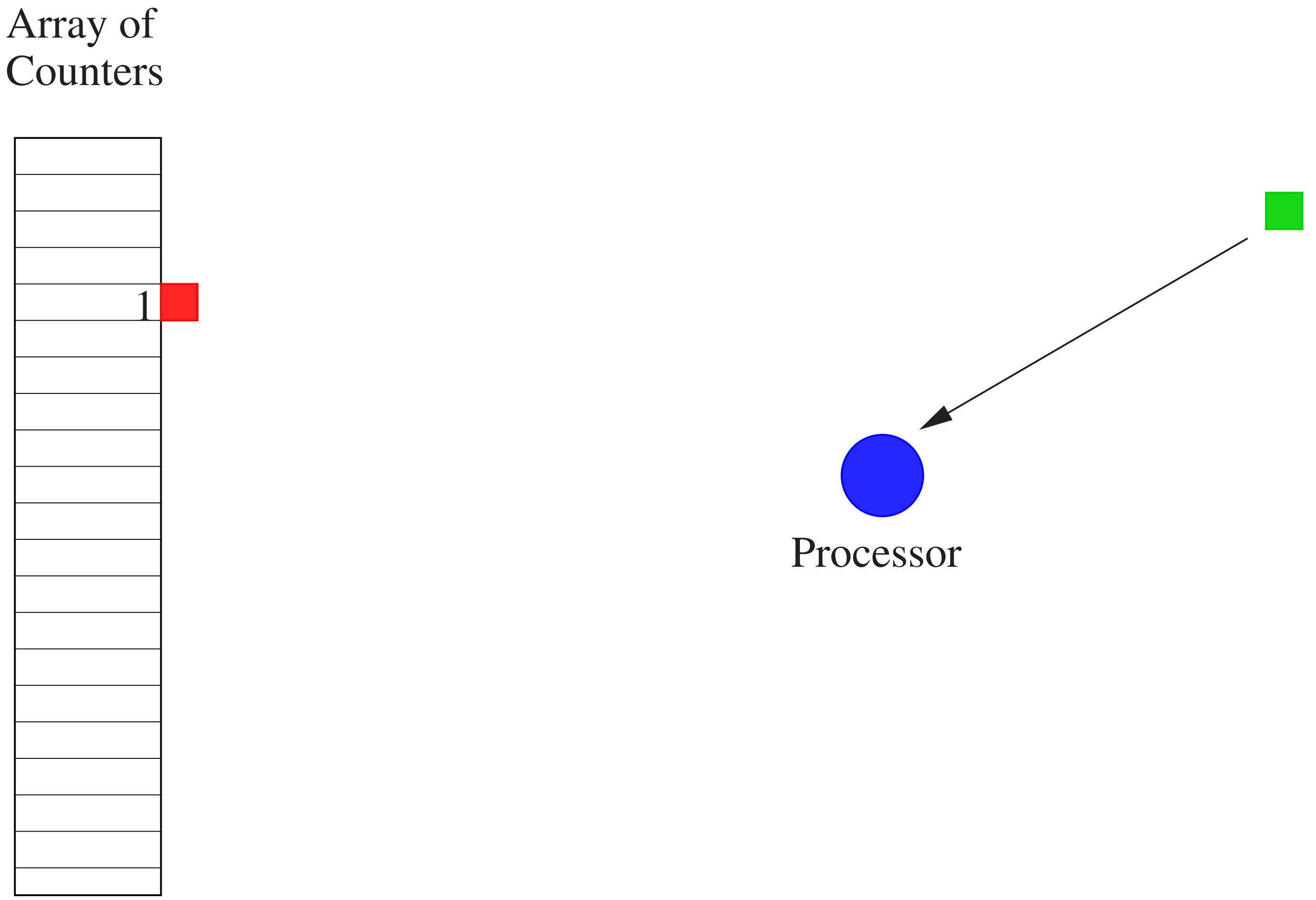


Counting Sketch: Array of counters

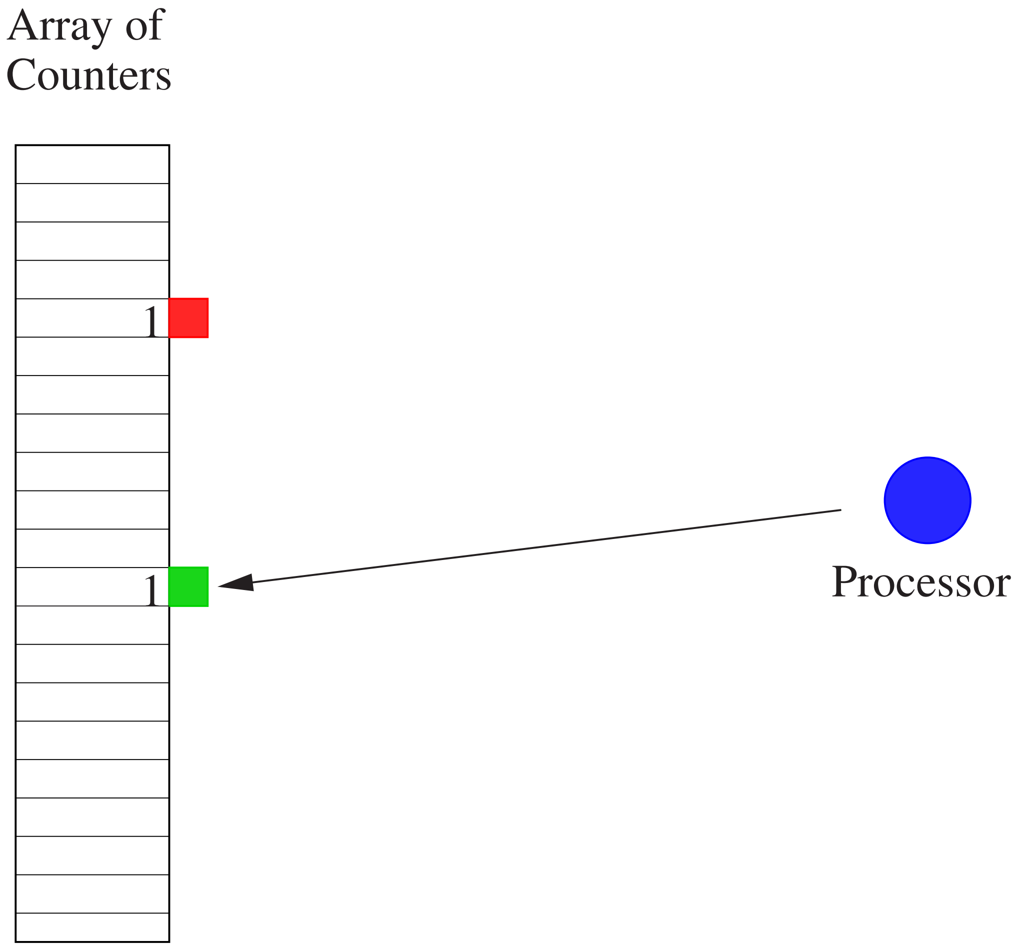
Array of
Counters



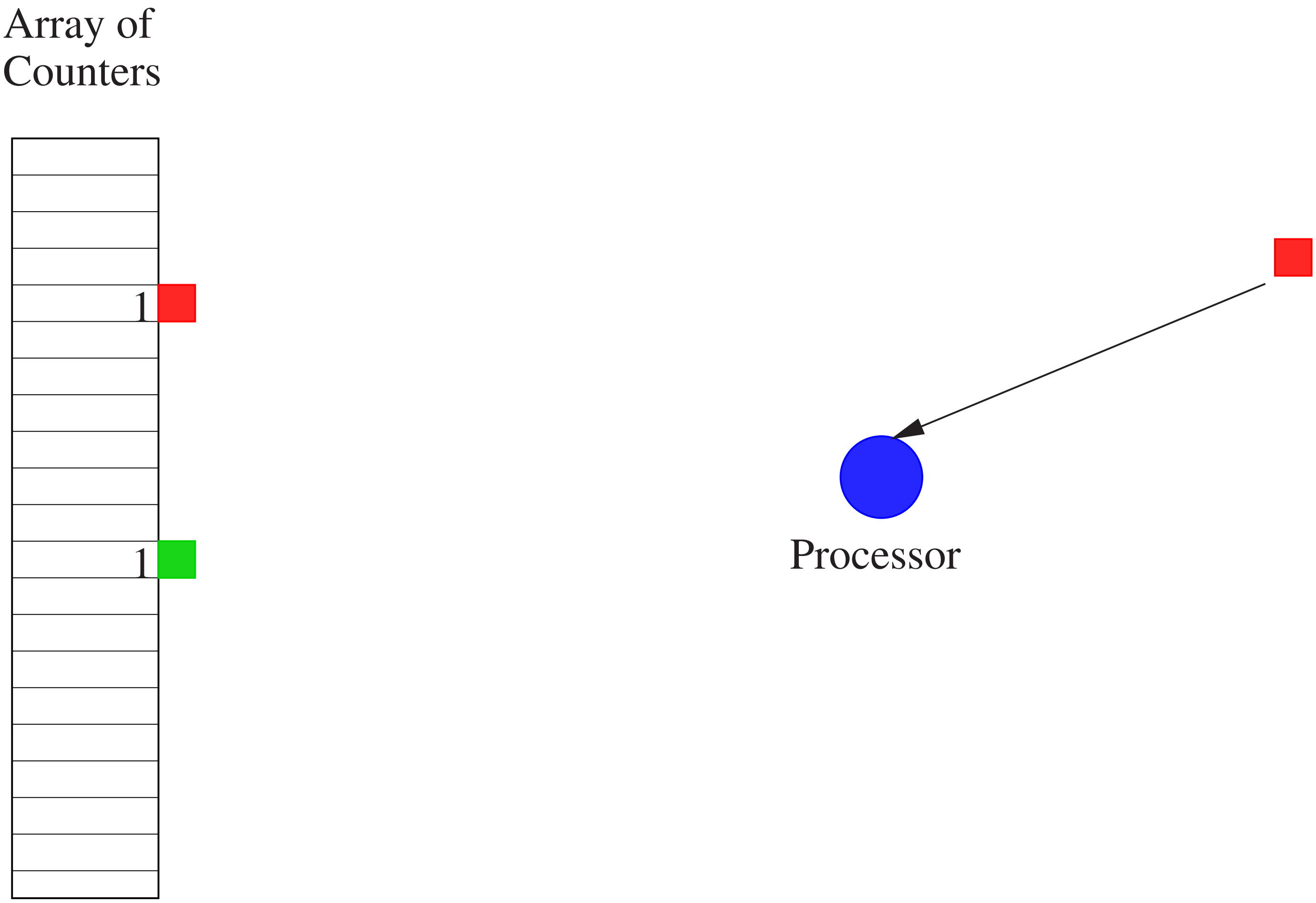
Counting Sketch: Array of counters



Counting Sketch: Array of counters

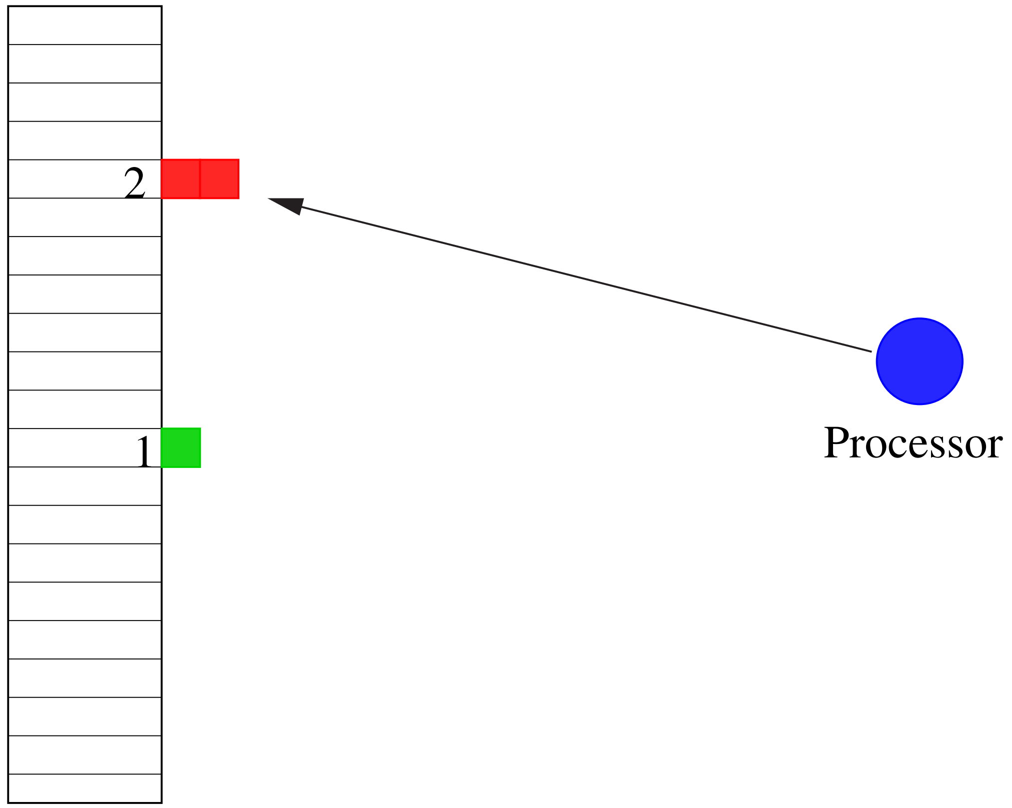


Counting Sketch: Array of counters

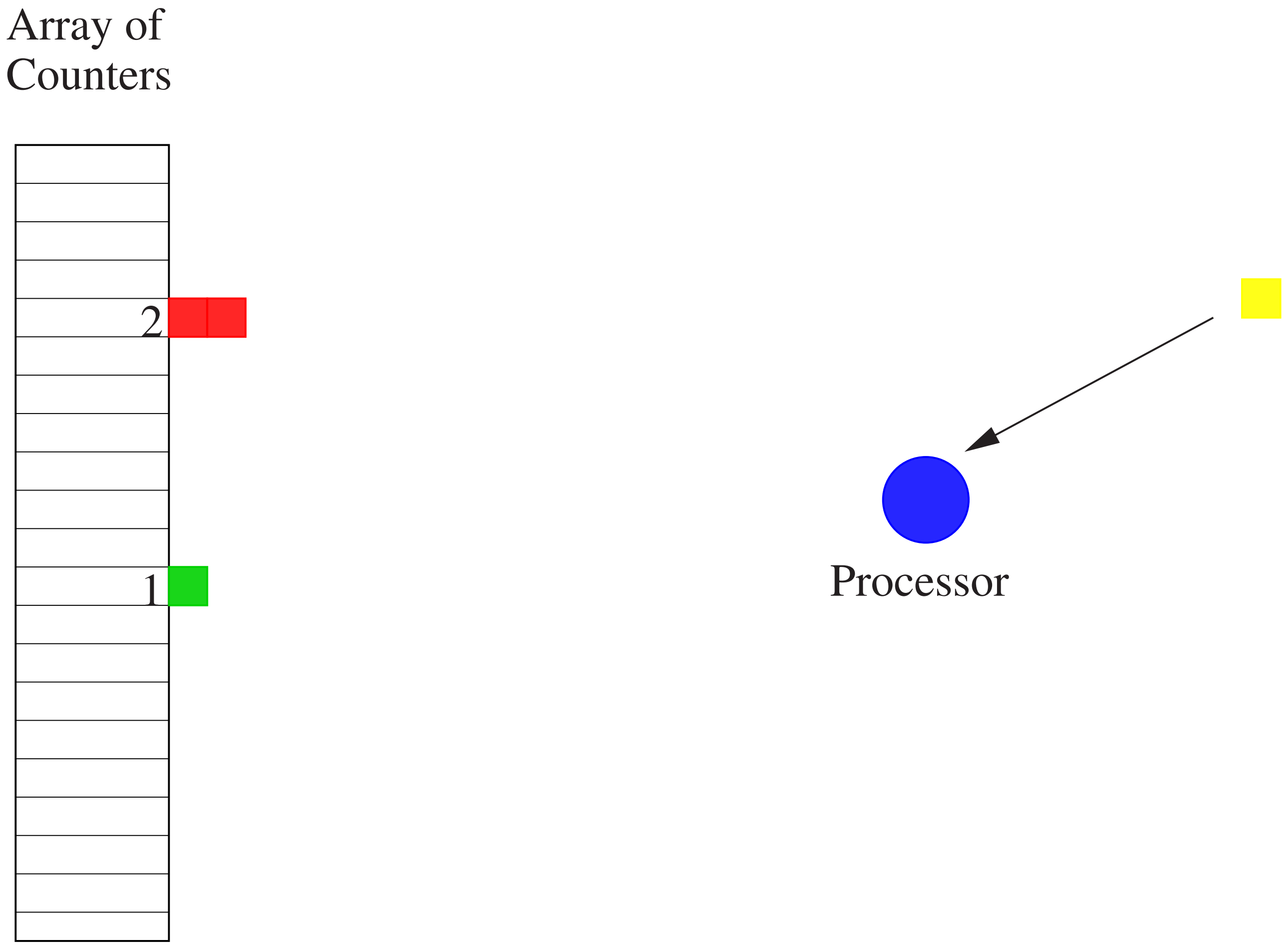


Counting Sketch: Array of counters

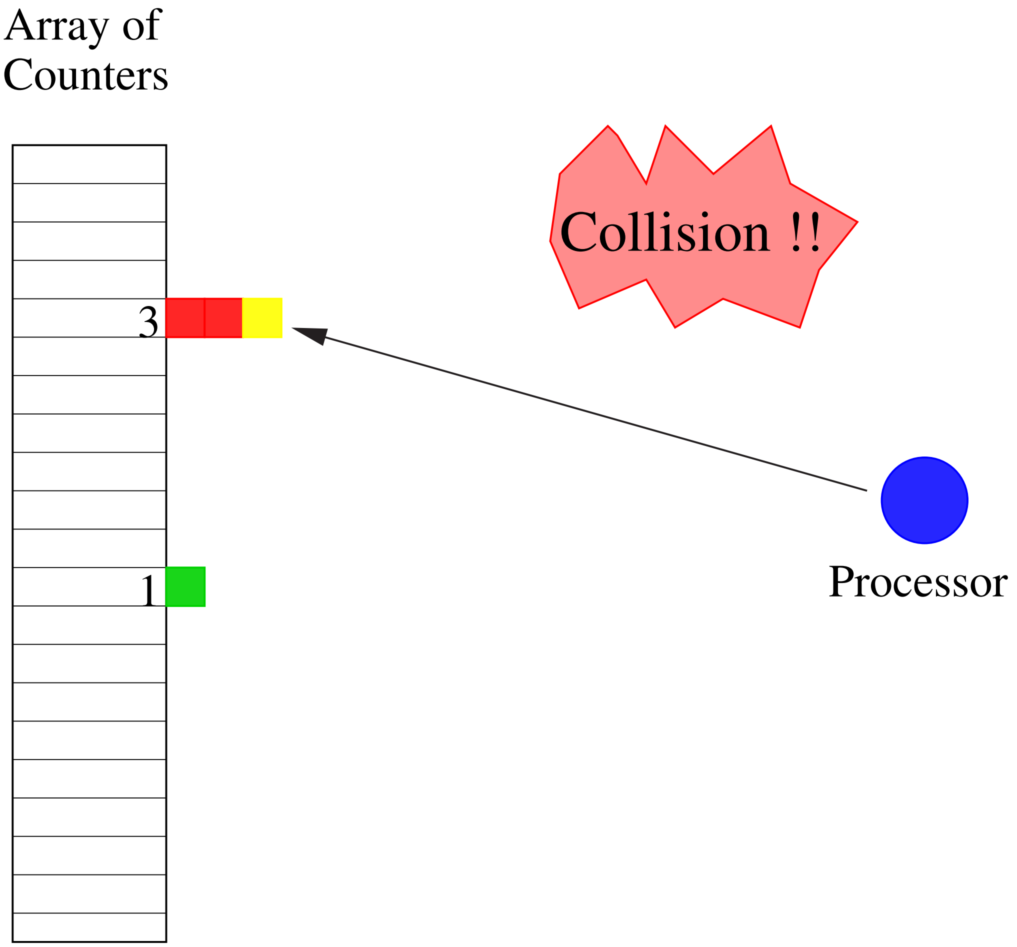
Array of
Counters



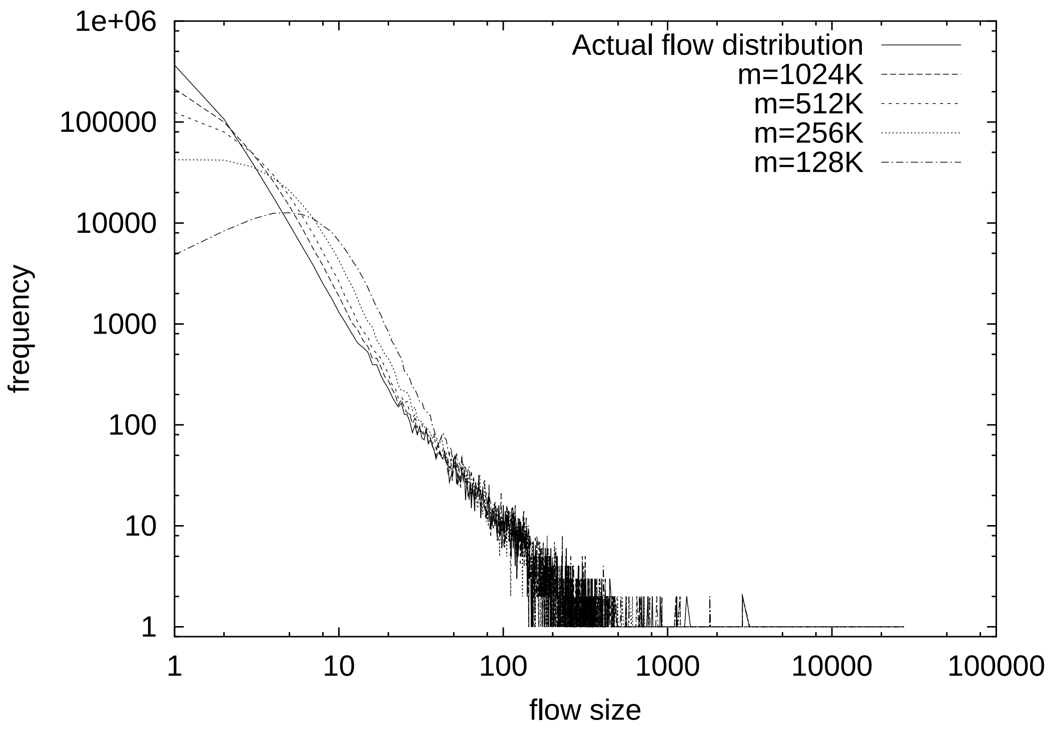
Counting Sketch: Array of counters



Counting Sketch: Array of counters



The shape of the "Counter Value Distribution"



The distribution of flow sizes and raw counter values (both x and y axes are in log-scale). $m =$ *number of counters*.

Estimating n and n_1

- Let total number of counters be m.
- Let the number of value-0 counters be m_0
- Then $\hat{n} = m * ln(m/m_0)$
- Let the number of value-1 counters be y_1
- Then $\hat{n_1} = y_1 e^{\hat{n}/m}$
- Generalizing this process to estimate n_2, n_3, and the whole flow size distribution will not work
- Solution: joint estimation using Expectation Maximization

Estimating the entire distribution, ϕ, using EM

- Begin with a guess of the flow distribution, ϕ^{ini}.
- Based on this ϕ^{ini}, compute the various possible ways of "splitting" a particular counter value and the respective probabilities of such events.
- This allows us to compute a refined estimate of the flow distribution ϕ^{new}.
- Repeating this multiple times allows the estimate to converge to a *local maximum*.
- This is an instance of *Expectation maximization*.

Estimating the entire flow distribution — an example

- For example, a counter value of 3 could be caused by three events:
  - 3 = 3 (no hash collision);
  - 3 = 1 + 2 (a flow of size 1 colliding with a flow of size 2);
  - 3 = 1 + 1 + 1 (three flows of size 1 hashed to the same location)
- Suppose the respective probabilities of these three events are 0.5, 0.3, and 0.2 respectively, and there are 1000 counters with value 3.
- Then we estimate that 500, 300, and 200 counters split in the three above ways, respectively.
- So we credit 300 * 1 + 200 * 3 = 900 to n_1, the count of size 1 flows, and credit 300 and 500 to n_2 and n_3, respectively.

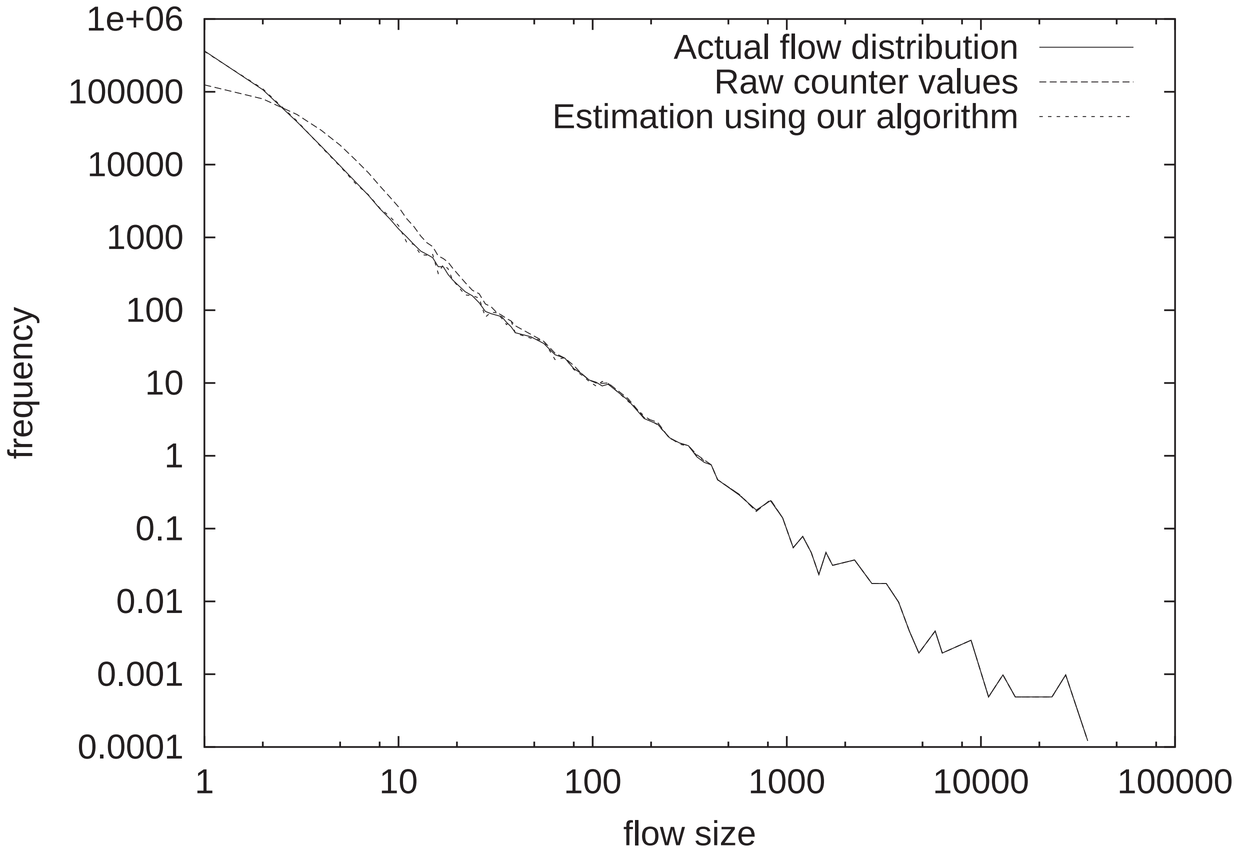
How to compute these probabilities

- Fix an arbitrary index ind. Let β be the event that f_1 flows of size s_1, f_2 flows of size s_2, ..., f_q flows of size s_q collide into slot ind, where $1 \leq s_1 < s_2 < \ldots < s_q \leq z$, let λ_i be n_i/m and λ be their total.
- Then, the a priori (i.e., before observing the value v at ind) probability that event β happens is

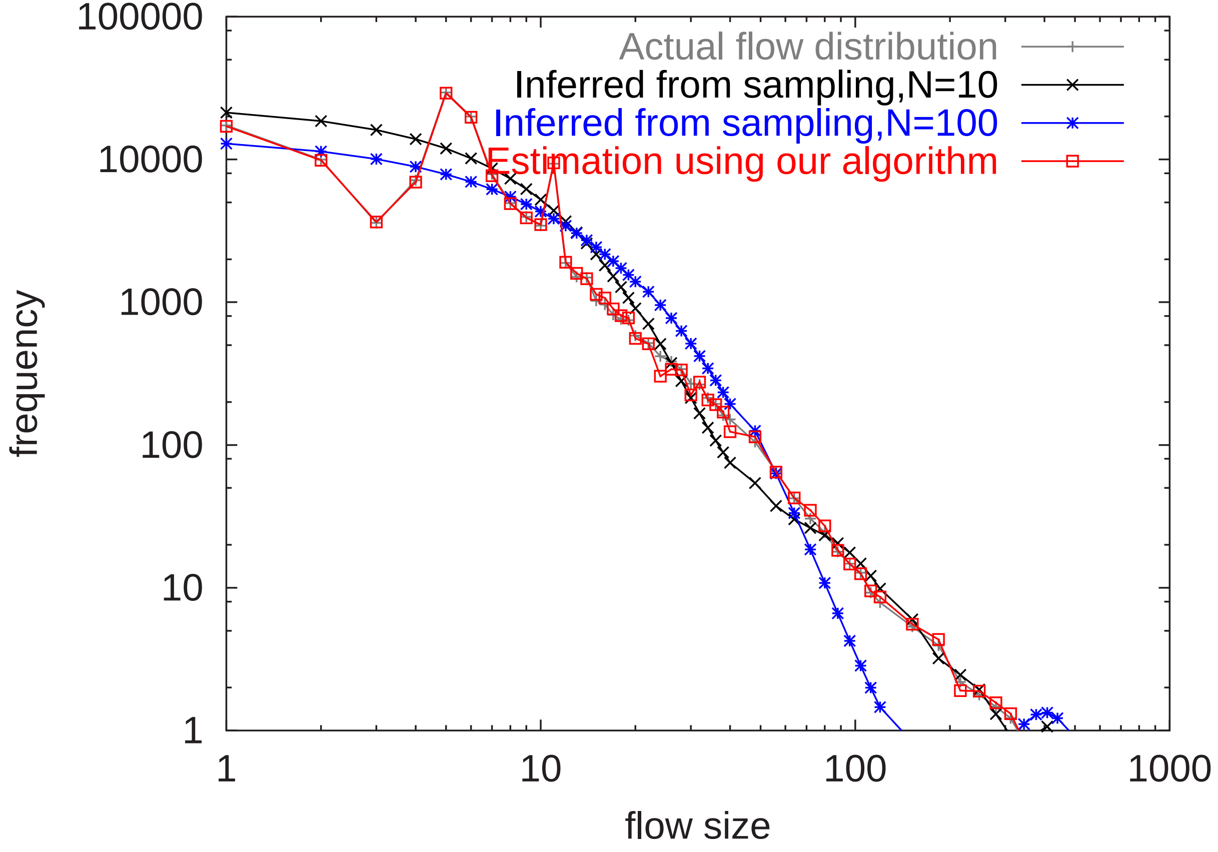
$$p(\beta|\phi, n) = e^{-\lambda} \prod_{i=1}^{q} \frac{\lambda_{s_i}^{f_i}}{f_i!}.$$

- Let Ω_v be the set of all collision patterns that add up to v. Then by Bayes' rule, $p(\beta|\phi, n, v) = \frac{p(\beta|\phi,n)}{\sum_{\alpha \in \Omega_v} p(\alpha|\phi,n)}$, where $p(\beta|\phi, n)$ and $p(\alpha|\phi, n)$ can be computed as above

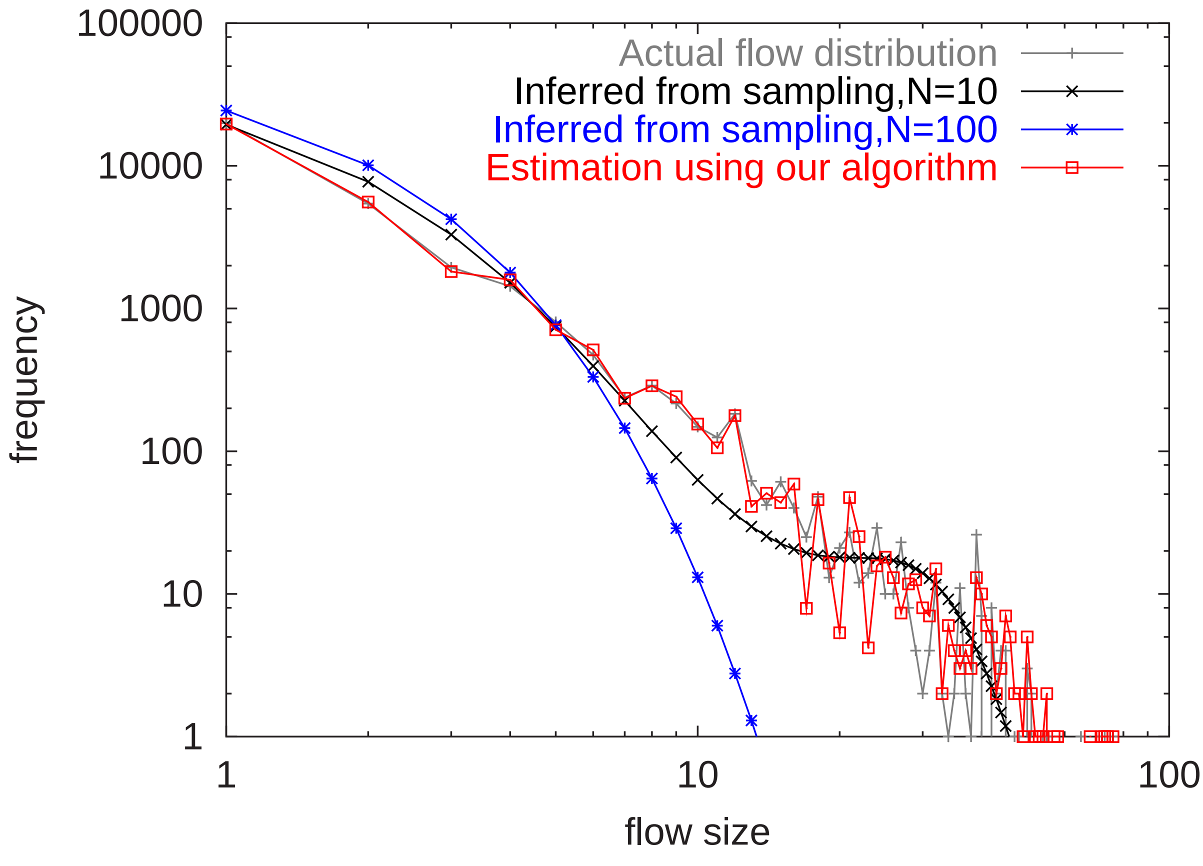
Evaluation — Before and after running the Estimation algorithm



Sampling vs. array of counters – Web traffic.



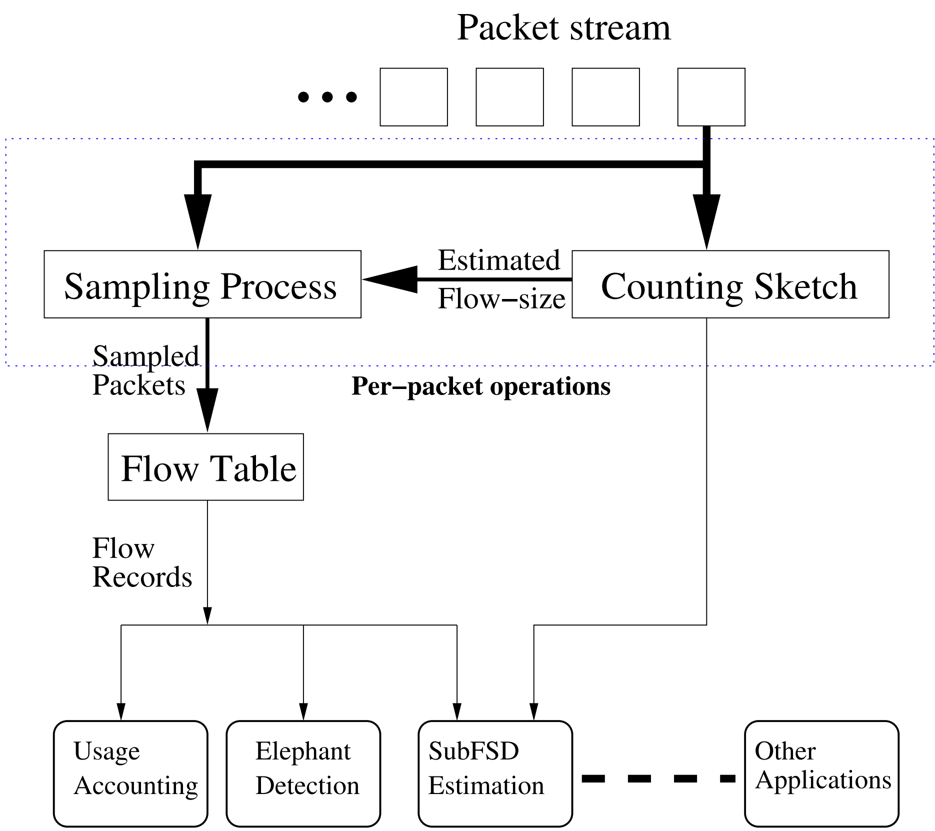
Sampling vs. array of counters – DNS traffic.



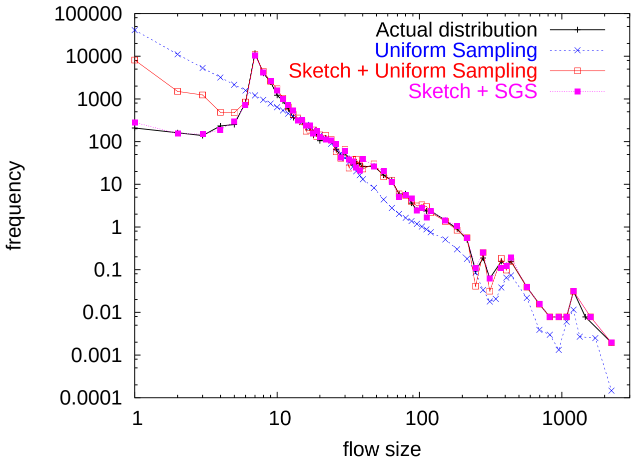
Extending the work to estimating subpopulation FSD [Sigmetrics 2005]

- Motivation: there is often a need to estimate the FSD of a subpopulation (e.g., “what is FSD of all the DNS traffic”).
- Definitions of subpopulation not known in advance and there can be a large number of potential subpopulations.
- Our scheme can estimate the FSD of any subpopulation defined after data collection.
- Main idea: perform both data streaming and sampling, and then correlate these two outputs (using EM).

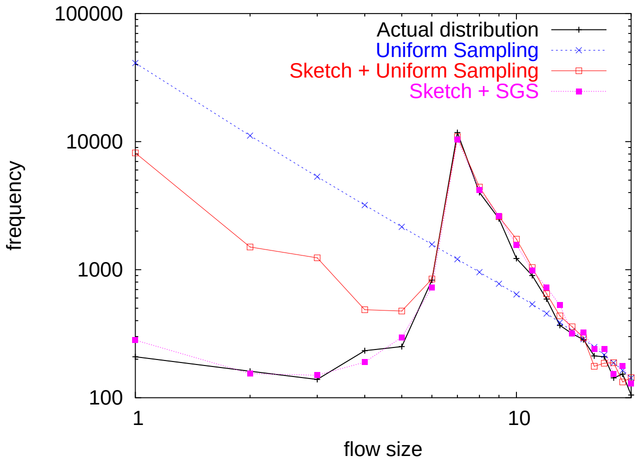
Streaming-guided sampling [Infocom 2006]



Estimating the Flow-size Distribution: Results



(a) Complete distribution.



(b) Zoom in to show impact on small flows.

Figure 1: Estimates of FSD of https flows using various data sources.

A hardware primitive for counter management (Sigmetrics 2006)

- Problem statement: To maintain a large array (say millions) of counters that need to be incremented (by 1) in an arbitrary fashion (i.e., $A[i_1] + +$, $A[i_2] + +$, ...)
- Increments may happen at very high speed (say one increment every 10ns) – has to use high-speed memory (SRAM)
- Values of some counters can be very large
- Fitting everything in an array of "long" (say 64-bit) SRAM counters can be expensive
- Possibly lack of locality in the index sequence (i.e., i_1, i_2, ...) – forget about caching

Motivations

- A key operation in many network data streaming algorithms is to “hash and increment”
- Routers may need to keep track of many different counts (say for different source/destination IP prefix pairs)
- To implement millions of token/leaky buckets on a router
- Extensible to other non-CS applications such as sewage management
- Our work is able to make 16 SRAM bits out of 1 (Alchemy of the 21st century)

Main Idea in Previous Approaches [SIPM:2001,RV:2003]

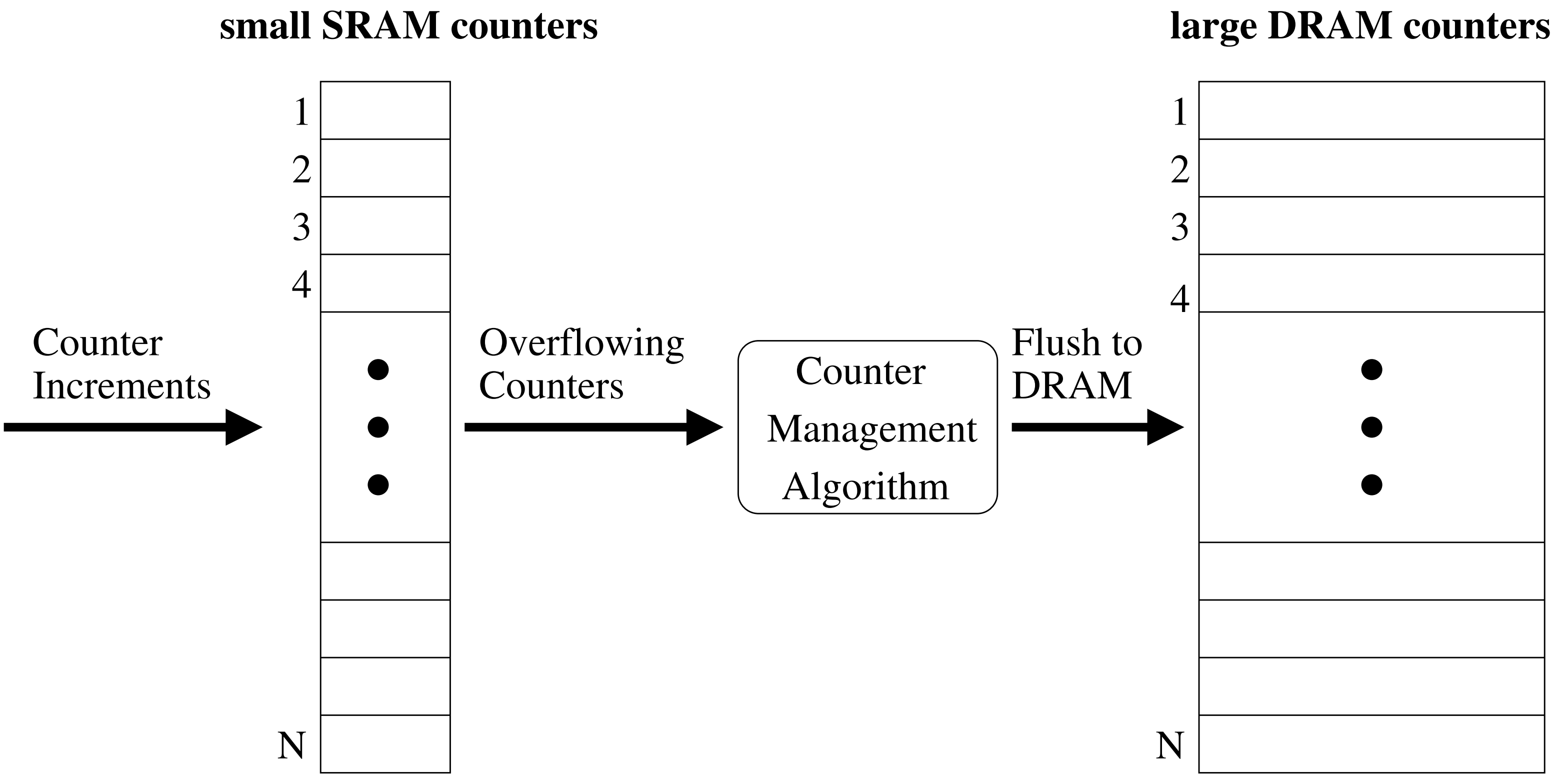


Figure 2: Hybrid SRAM/DRAM counter architecture

CMA used in [SIPM:2001]

- D. Shah, S. Iyer, B. Prabhakar, and N. McKeown, "Maintaining statistics counters in router line cards", *Hot Interconnects 2001*
- Implemented as a priority queue (fullest counter first)
- Need 28 = 8 + 20 bits per counter (when S/D is 12) – the theoretical minimum is 4
- Need pipelined hardware implementation of a heap.

CMA used in [RV:2003]

- S. Ramabhadran and G. Varghese, “Efficient implementation of a statistics counter architecture”, *ACM SIGMETRICS 2003*
- SRAM counters are tagged when they are at least half full (implemented as a bitmap)
- Scan the bitmap clockwise (for the next “1”) to flush (half-full)$^{+}$ SRAM counters, and pipelined hierarchical data structure to “jump to the next 1” in O(1) time
- Maintain a small priority queue to preemptively flush the SRAM counters that rapidly become completely full
- 8 SRAM bits per counter for storage and 2 bits per counter for the bitmap control logic, when S/D is 12.

Our scheme

- Our scheme only needs 4 SRAM bits when S/D is 12.
- Flush only when an SRAM counter is “completely full” (e.g., when the SRAM counter value changes from 15 to 16 assuming 4-bit SRAM counters).
- Use a small (say hundreds of entries) SRAM FIFO buffer to hold the indices of counters to be flushed to DRAM
- Key innovation: a simple randomized algorithm to ensure that counters do not overflow in a burst large enough to overflow the FIFO buffer, with overwhelming probability
- Our scheme is provably space-optimal

The randomized algorithm

- Set the initial values of the SRAM counters to independent random variables uniformly distributed in $\{0, 1, 2, ..., 15\}$ (i.e., $A[i] := uniform\{0, 1, 2, ..., 15\}$).
- Set the initial value of the corresponding DRAM counter to the negative of the initial SRAM counter value (i.e., $B[i] := -A[i]$).
- Adversaries know our randomization scheme, but not the initial values of the SRAM counters
- We prove rigorously that a small FIFO queue can ensure that the queue overflows with very small probability

A numeric example

- One million 4-bit SRAM counters (512 KB) and 64-bit DRAM counters with SRAM/DRAM speed difference of 12
- 300 slots ($\approx$ 1 KB) in the FIFO queue for storing indices to be flushed
- After 10^{12} counter increments in an arbitrary fashion (like 8 hours for monitoring 40M packets per second links)
- The probability of overflowing from the FIFO queue: less than 10^{-14} in the worst case (MTBF is about 100 billion years) – proven using minimax analysis and large deviation theory (including a new tail bound theorem)

Finding Global Icebergs over Distributed Data Sets (PODS 2006)

- An **iceberg**: the item whose frequency count is greater than a certain threshold.
- A number of algorithms are proposed to find icebergs at a single node (i.e., local icebergs).
- In many real-life applications, data sets are physically distributed over a large number of nodes. It is often useful to find the icebergs over aggregate data across all the nodes (i.e., **global icebergs**).
- Global iceberg $\neq$ Local iceberg
- We study the problem of finding global icebergs over distributed nodes and propose two novel solutions.

Motivations: Some Example Applications

- Detection of distributed DoS attacks in a large-scale network
  - The IP address of the victim appears over many ingress points. It may not be a local iceberg at any ingress points since the attacking packets may come from a large number of hosts and Internet paths.
- Finding globally frequently accessed objects/URLs in CDNs (e.g., Akamai) to keep tabs on current “hot spots”
- Detection of system events which happen frequently across the network during a time interval
  - These events are often the indication of some anomalies. For example, finding DLLs which have been modified on a large number of hosts may help detect the spread of some unknown worms or spywares.

Problem statement

- A system or network that consists of N distributed nodes
- The data set S_i at node i contains a set of $\langle x, c_{x,i} \rangle$ pairs.
  - Assume each node has enough capacity to process incoming data stream. Hence each node generates a list of the arriving items and their exact frequency counts.
- The flat communication infrastructure, in which each node only needs to communicate with a central server.
- Objective: Find $\{x | \sum_{i=1}^{N} c_{x,i} \geq T\}$, where $c_{x,i}$ is the frequency count of the item x in the set S_i, with the minimal communication cost.

Our solutions and their impact

- Existing solutions can be viewed as “hard-decision codes” by finding and merging local icebergs
- We are the first to take the “soft-decision coding” approach to this problem: encoding the “potential” of an object to become a global iceberg, which can be decoded with overwhelming probability if indeed a global icerbeg
- Equivalent to the minimax problem of “corrupted politician”
- We offered two solution approaches (sampling-based and bloom-filter-based)and discovered the beautiful mathematical structure underneath (discovered a new tail bound theory on the way)
- Sprint, Thompson, and IBM are all very interested in it

A New Tail Bound Theorem

- Given any $\theta > 0$ and $\epsilon > 0$, the following holds: Let $W_j, 1 \leq j \leq m$, m arbitrary, be independent random variables with $EXP[W_j] = 0$, $|W_j| \leq \theta$ and $VAR[W_j] = \sigma_j^2$. Let $W = \sum_{j=1}^{m} W_j$ and $\sigma^2 = \sum_{i=1}^{m} \sigma_j^2$ so that $VAR[W] = \sigma^2$. Let $\delta = \ln(1+\epsilon)/\theta$. Then for $0 < a \leq \delta\sigma$,

$$\Pr[W > a\sigma] < e^{-\frac{a^2}{2}(1-\frac{\epsilon}{3})}$$

- This theorem is used in both the counter array work and the global iceberg work.

Conclusions

- Data streaming can take the forms of SSSG, MSSG, and MSMG.
- We presented a SSSG data streaming algorithm trio and a hardware primitive to support all of them (and some other algorithms).
- A quick sampler of a MSSG algorithm: distributed iceberg query