Compact algorithms for measuring network performance

Yufei Zheng

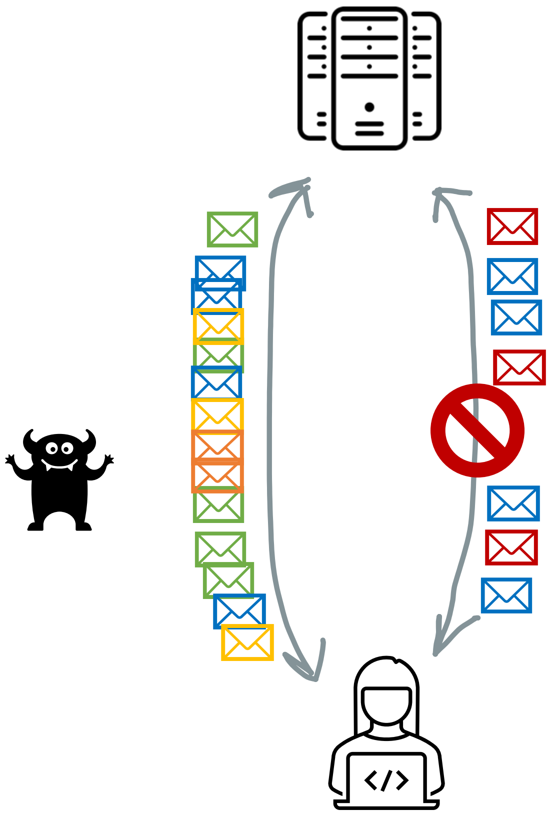
Examiners: Jennifer Rexford (adviser), Maria Apostolaki, and Mark Braverman

Readers: Huacheng Yu and David Hay

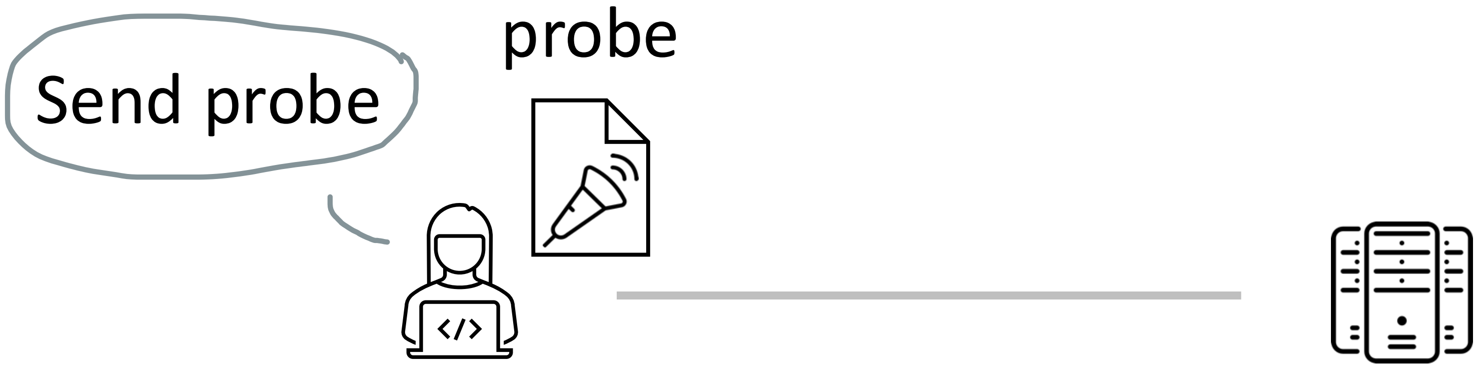
FPO August 22, 2024

Performance monitoring is important

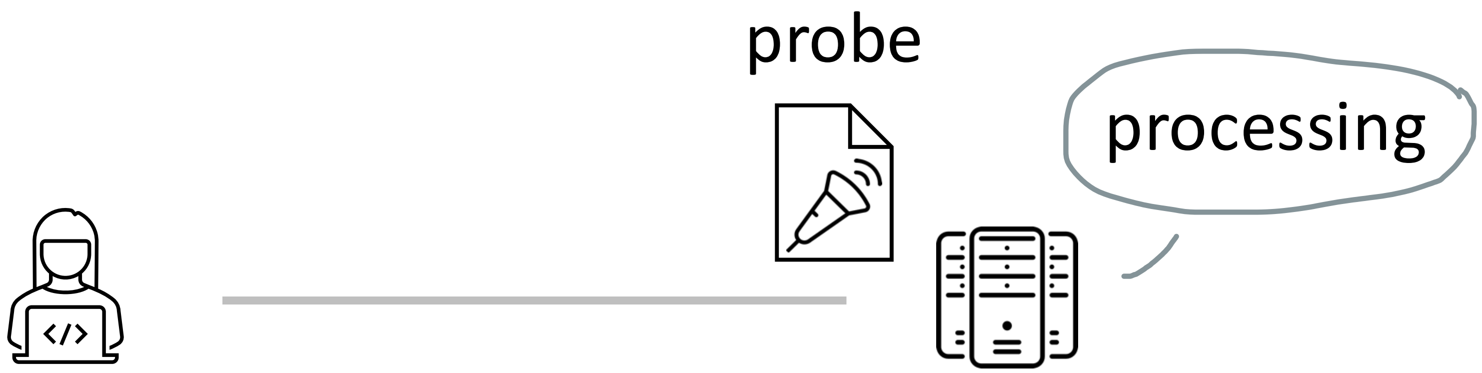
- Identify bottlenecks and latency issues
  - Optimize network for peak efficiency
- Pinpoint congested paths
- Identify security threats
  - Route traffic away from malicious paths



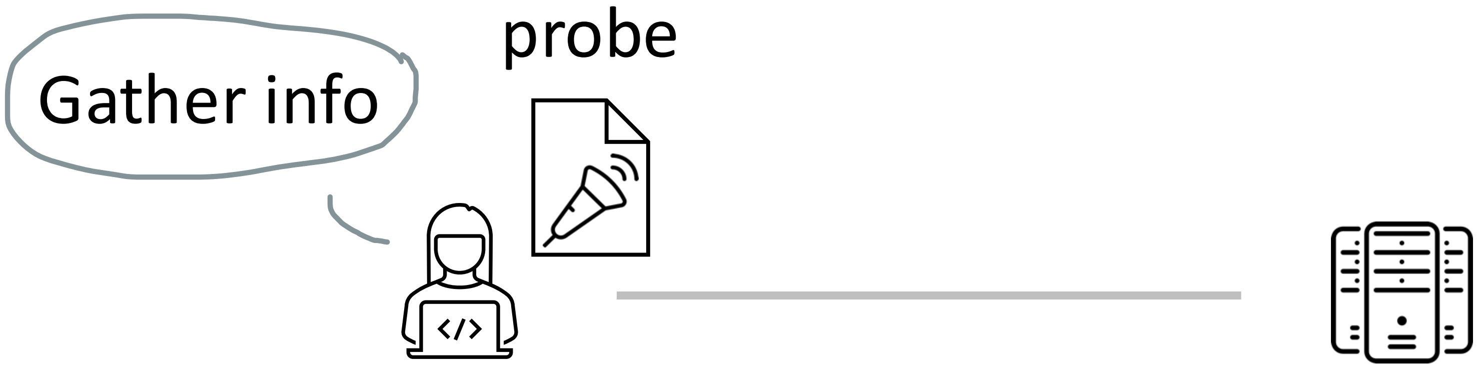
Traditional performance monitoring: ***Active probing***



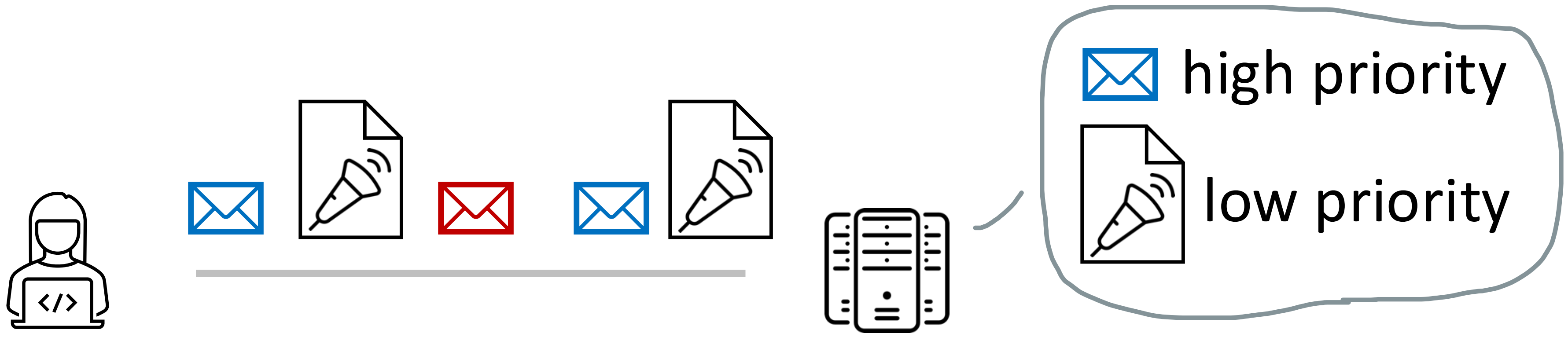
Traditional performance monitoring: ***Active probing***



Traditional performance monitoring: ***Active probing***



Traditional performance monitoring: *Active probing*

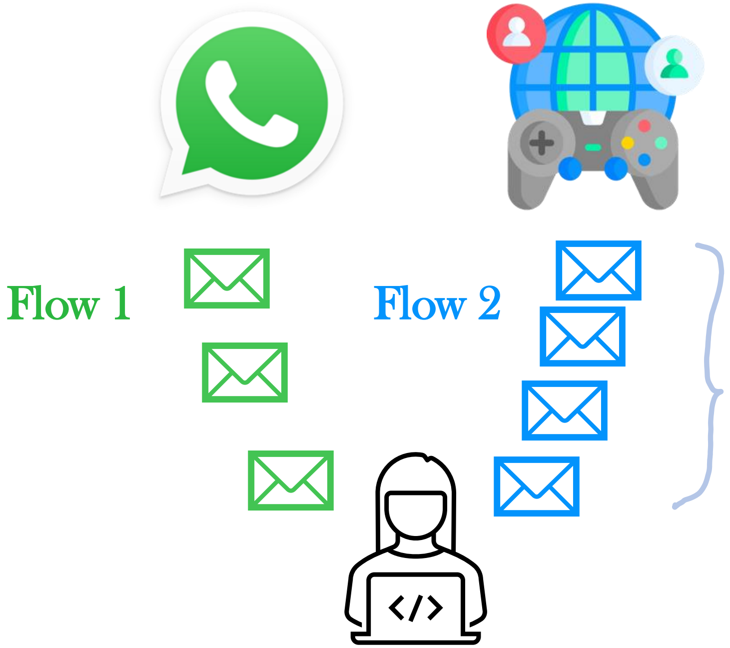


- Probes add excessive traffic
- Performance of probes not necessarily representative
- Access network or end-host issues may dominate

Passive monitoring – analyzing existing traffic

- No impact on the performance of a network
- Realistic view of network utilization and congestion

A *flow* refers to a sequence of packets that belong to a communication session



Share common attributes:

src and dst IPs
src and dst ports
protocol type

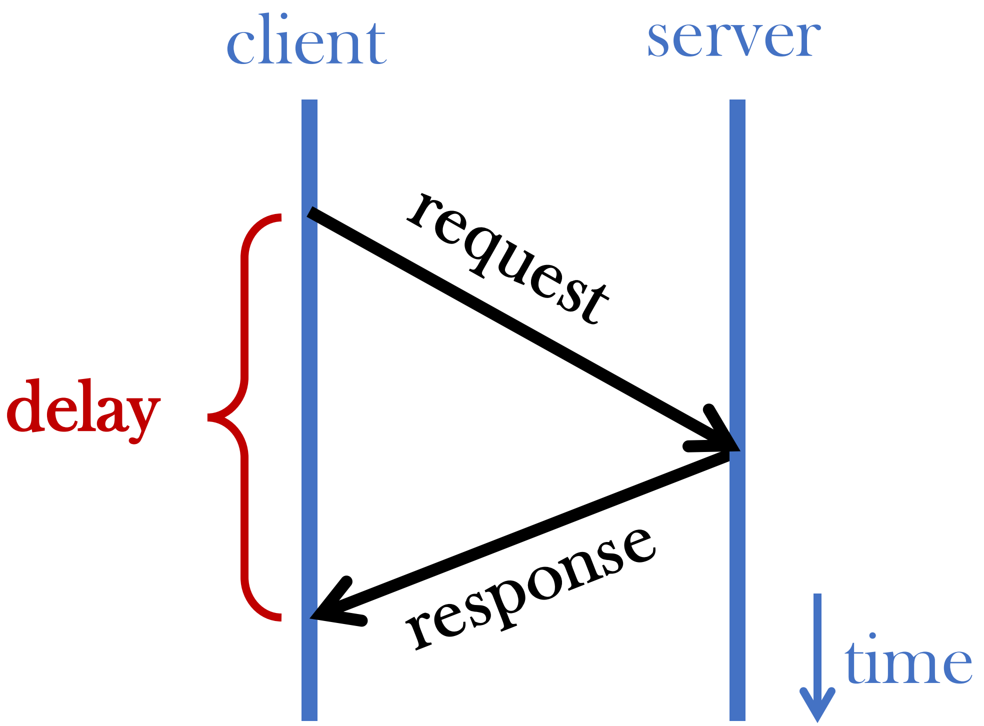
To measure network performance, each packet must be processed ***in conjunction with its predecessor*** in the same flow.

We can focus on TCP

- TCP packets contain crucial information (src & dst ports, SEQ#, ACK#, flags)
  - Makes it possible to identify flows and compute metrics
  - Provides context for the network behavior observed
- Widely used, accounts for the vast majority of network traffic

Performance metrics

Round-trip delay



TCP packet reordering

Programmable data plane

Flexible parsing ⇒ Extract header fields

Arrays ⇒ Keep state across successive packets of the same flow

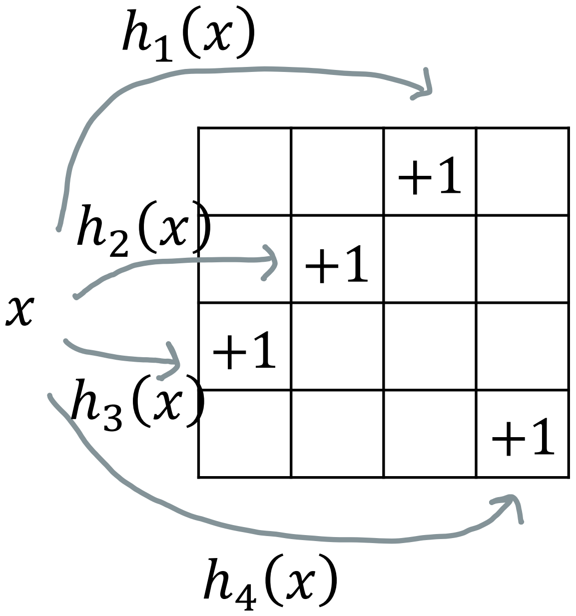
Simple arithmetic operations ⇒ Compute time differences, tally counts

Data plane restrictions

- Limited amount of memory compared to the #concurrent flows
- Can only access memory a few times per packet
- Tasks share limited memory resources
- Limited bandwidth for communicating with control plane

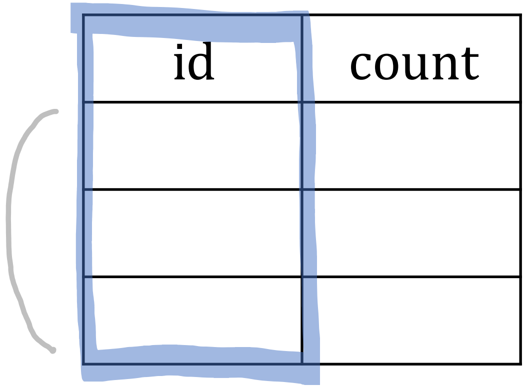
Previous work on measuring ***volume-based*** metrics

Sketch-based algos



Counter-based algos

subset of traffic



Match packets
from the same flow

$\Rightarrow$

performance metrics
for ***heavy*** flows

Lack performance guarantees!

We also need new algorithms

Also not always enough to focus on heavy flows

- **TCP packet reordering:** overlook congestion on paths with no heavy flow
- **Delay monitoring:** no notion of heaviness

Thesis outline

§ 2: Unbiased delay measurement

§ 3: TCP packet reordering

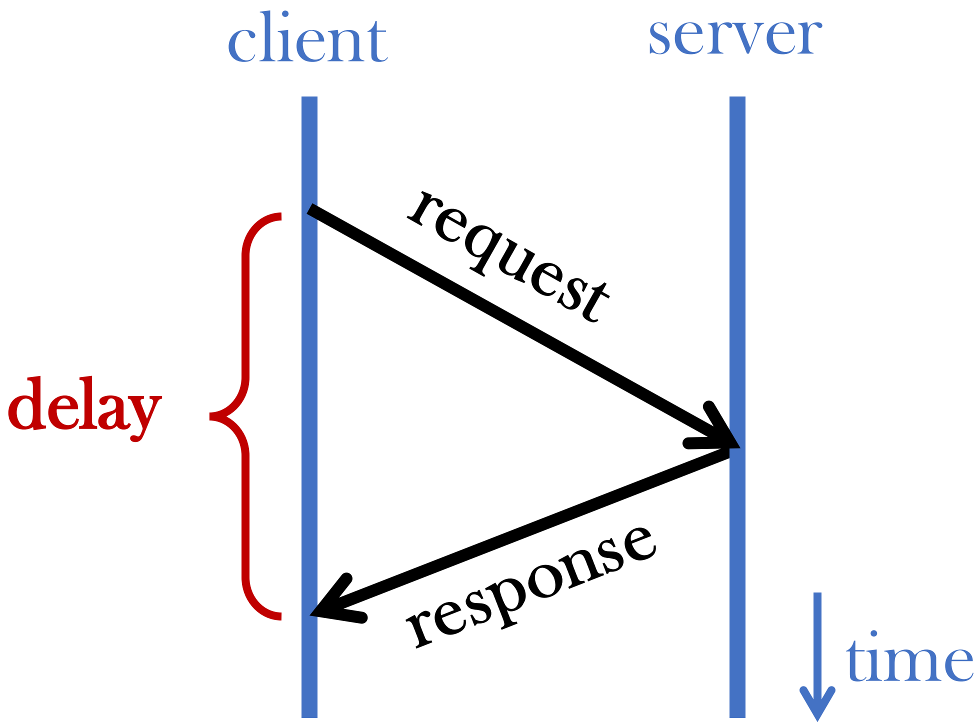
§ 4: Analysis of Random Admission Policy

What is delay

packets: requests, responses

>5% of the round-trip delays exceeds 50ms?

Client & ISP



Problem statement

Input

A stream of $(ID, \text{ts}, req/resp)$

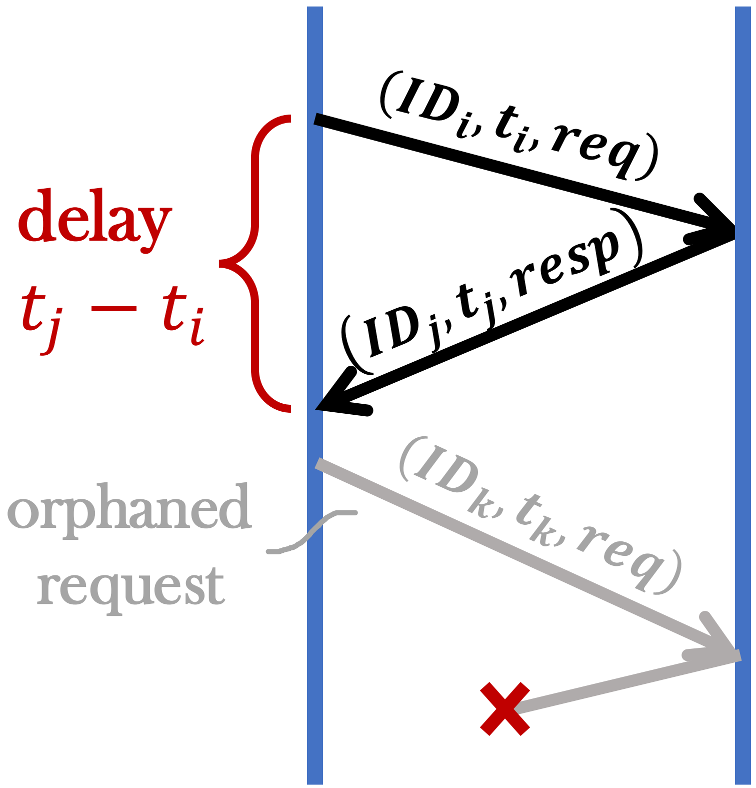
Data-plane algorithm

Output

An approx $\hat{F}(t)$ of ground truth delay CDF $F(t)$

Assumptions

- Each request has
  - Unique ID
  - ≤ 1 matching response
- Each matching pair of packets i, j
  - $ID_i = ID_j$
  - Delay $t_j - t_i$



A simple algo - inserting requests

Hash-indexed array

$$(\boldsymbol{ID_i}, \boldsymbol{t_i}, \boldsymbol{req}) \xrightarrow{\boldsymbol{h(ID_i)}}$$

ID	Timestamp
$\boldsymbol{ID_i}$	$\boldsymbol{t_i}$
$\boldsymbol{ID_k}$	$\boldsymbol{t_k}$

Insert to an empty slot

Chen, Kim, Aman, Chang, Lee & Rexford'20

A simple algo - inserting requests

(ID_j, t_j, req) $h(ID_j)$

Hash-indexed array

ID	Timestamp
ID_i	t_i
ID_j	t_j

Overwrite existing record (evict and insert)

Chen, Kim, Aman, Chang, Lee & Rexford'20

A simple algo - inserting requests

$(\boldsymbol{ID_i}, \boldsymbol{t_i}', \boldsymbol{req}) \xrightarrow{\boldsymbol{h(ID_i)}}$

Hash-indexed array

ID	Timestamp
ID_i	t_i
ID_j	t_j

Delay samples
$t_i' - t_i$

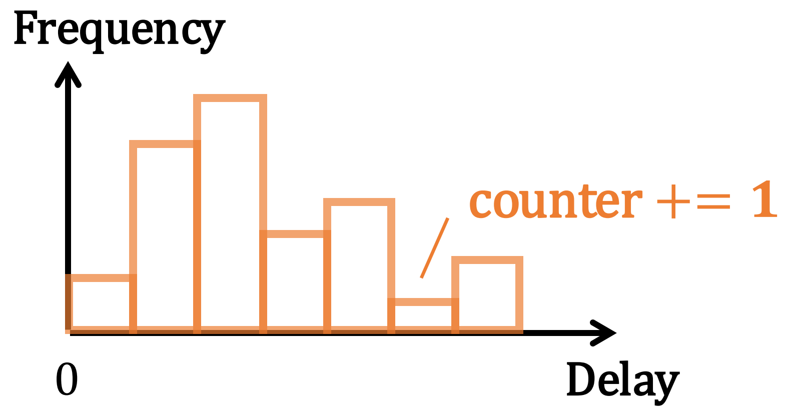
Match!
Report $t_i' - t_i$

Chen, Kim, Aman, Chang, Lee & Rexford'20

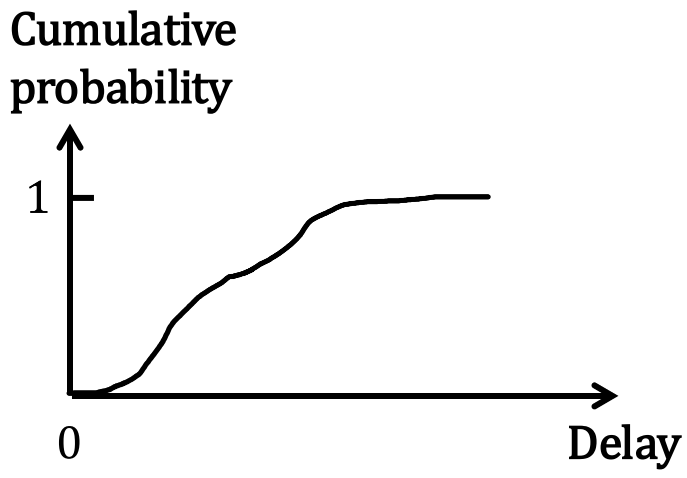
A simple algo – generating CDF

Delay samples
t_1
t_2
t_3
t_4
t_5

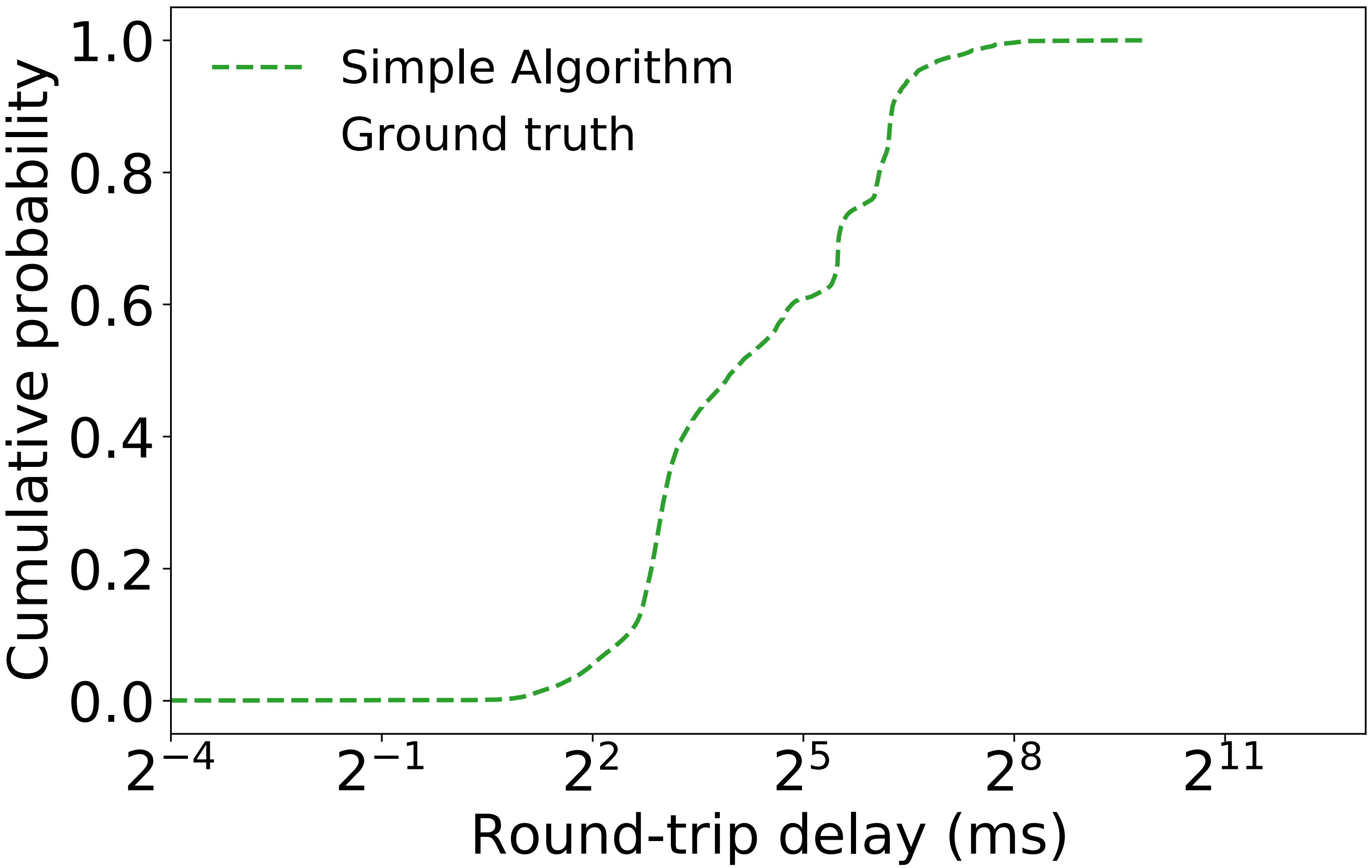
Frequency estimators (PDF)



CDF



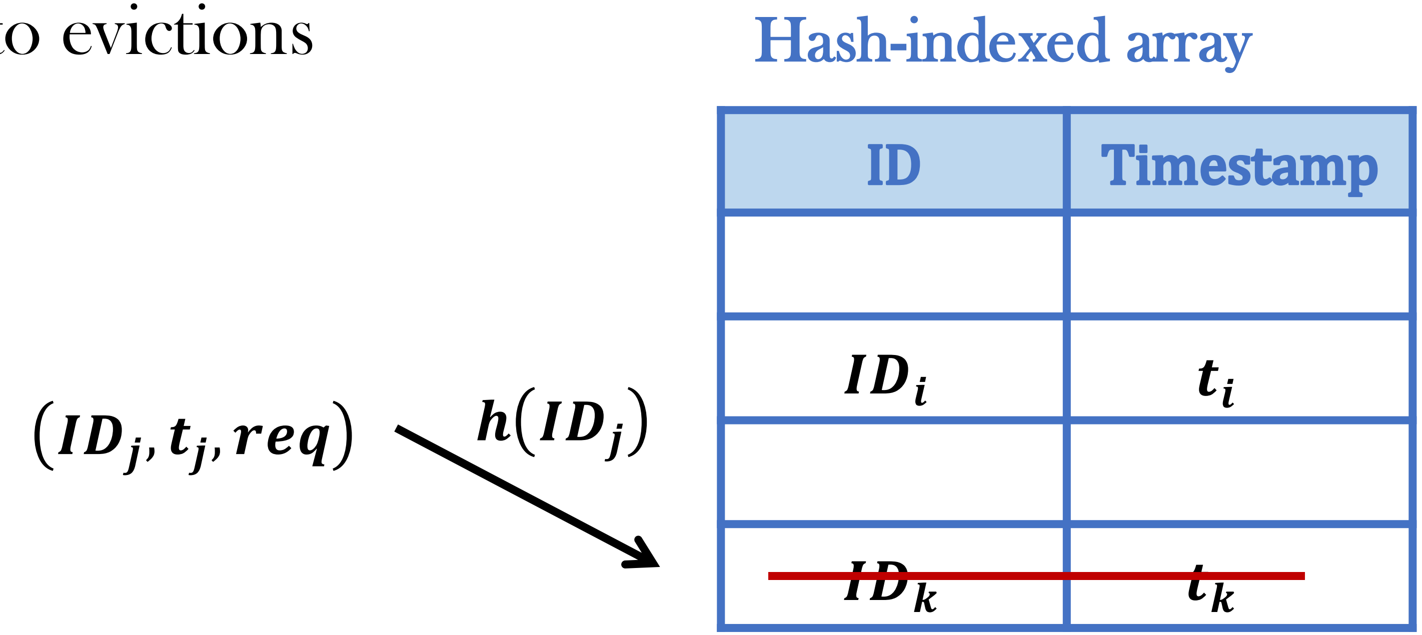
Bias against large delays



Bi-directional campus trace, memory size $M = 2^{16}$

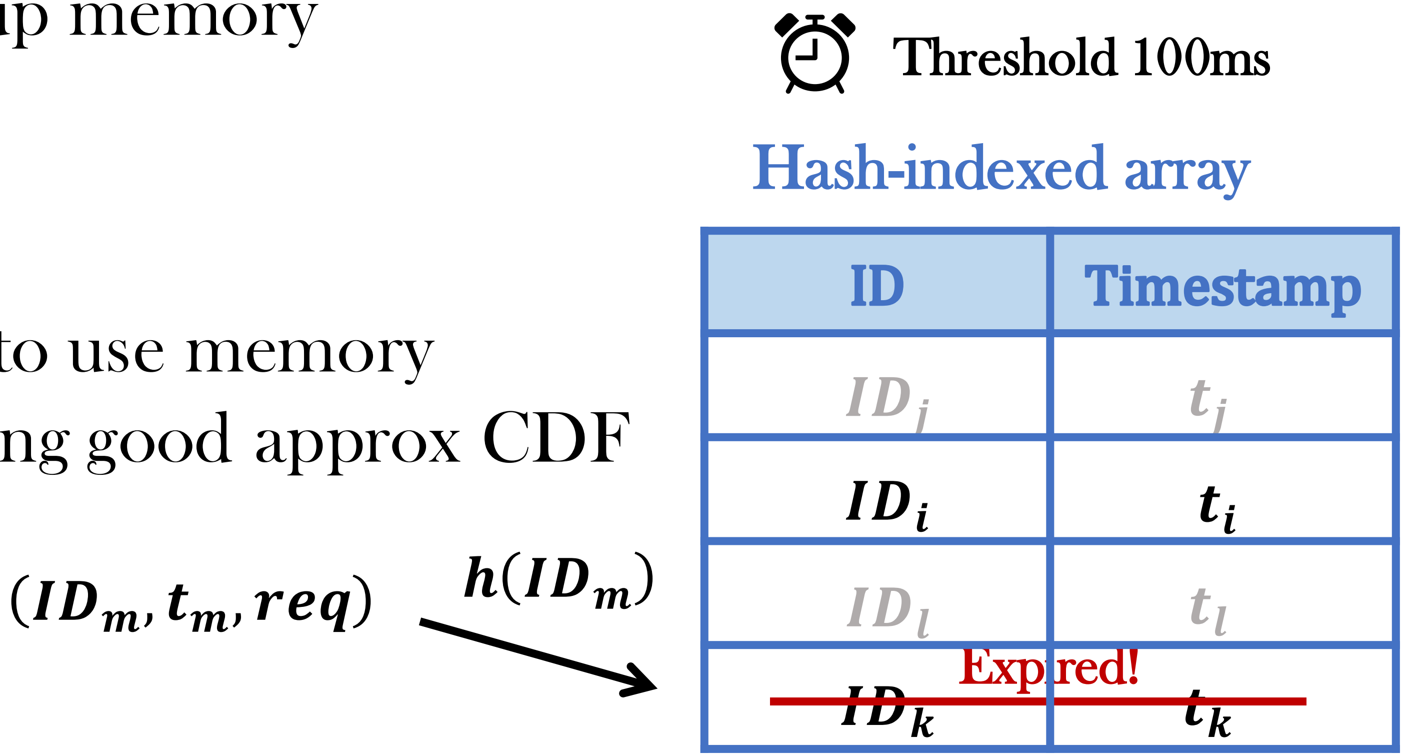
Survivorship bias against large delays

- **Why hard to sample a higher delay?**
  - Request stays in memory longer
  - Records in memory overwritten on hash collisions
  - More vulnerable to evictions



Attempts to mitigate bias

- Favor existing entries in memory
  - Orphaned requests fill up memory
  - Few new samples
- Use time threshold
  - Hard to tune threshold to use memory efficiently while generating good approx CDF



Main idea – correction factor

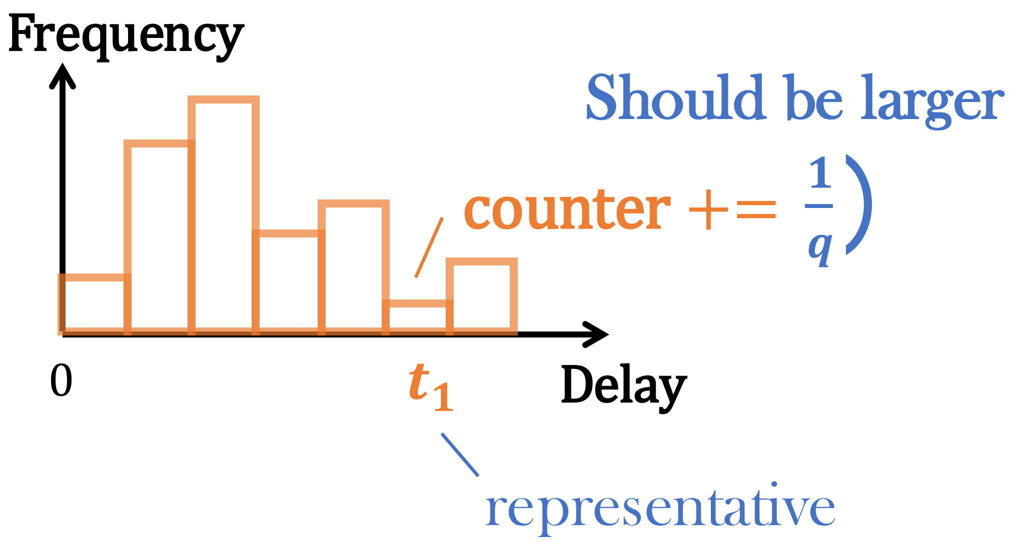
Overwrite on hash collisions + correct for bias

sampled w.p. q

$\Downarrow$ unbiased frequency

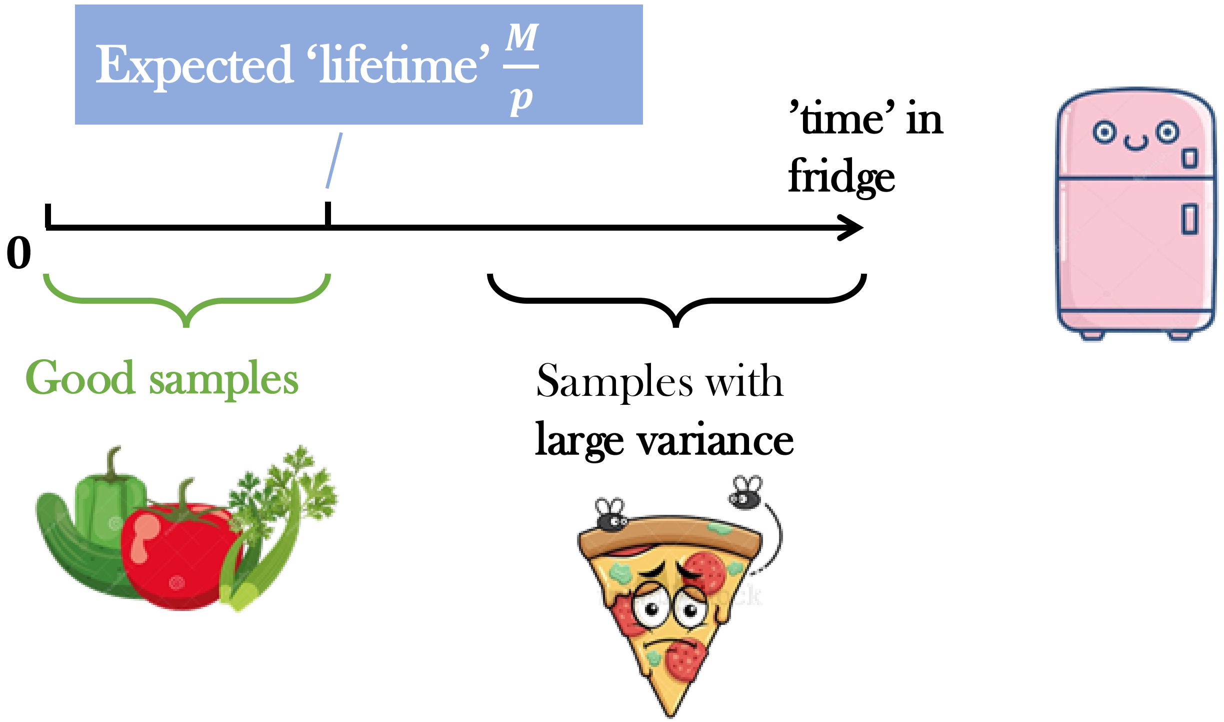
count as $\frac{1}{q}$ samples

Correction factor



Main idea – fridge-$(\boldsymbol{M}, \boldsymbol{p})$

a hash-indexed array of size M with entering probability p



Computing correction factor

a sample that survives $\boldsymbol{x}$ number of insertions between its $\boldsymbol{req}$ and $\boldsymbol{resp}$

Independent events: (1) Its request enters fridge (2) The x insertions into the fridge do not collide with its record

$$\mathbb{P}\left[\begin{matrix}\text{collecting a sample that} \\ \text{survives } x \text{ insertions}\end{matrix}\right] = \boldsymbol{p} \cdot \left(\mathbf{1} - \frac{\boldsymbol{p}}{\boldsymbol{M}}\right)^{\boldsymbol{x}}$$

$$\Longrightarrow \text{Correction factor} = \boldsymbol{p}^{-\mathbf{1}}\left(\mathbf{1} - \frac{\boldsymbol{p}}{\boldsymbol{M}}\right)^{-\boldsymbol{x}}$$

Single-fridge algo – generating reports

Fridge-(M, p)

ID	Timestamp	Insertion count
ID_i	t_i	x_i
ID_j	t_j	x_j

(ID_j, t_j, req)

$h(ID_j)$

Global insertion counter $+1$ $= x_j$

Single-fridge algo – generating reports

$(ID_i, t_i', resp) \rightarrow$ $h(ID_i)$

Fridge-(M, p)

ID	Timestamp	Insertion count
ID_i	t_i	x_i
ID_j	t_j	x_j

Delay samples	Correction factors
$t_i' - t_i$	$p^{-1}\left(1 - \frac{p}{M}\right)^{-(x_j - x_i)}$

Global insertion counter $= x_j$

Survives $x_j - x_i$ insertions

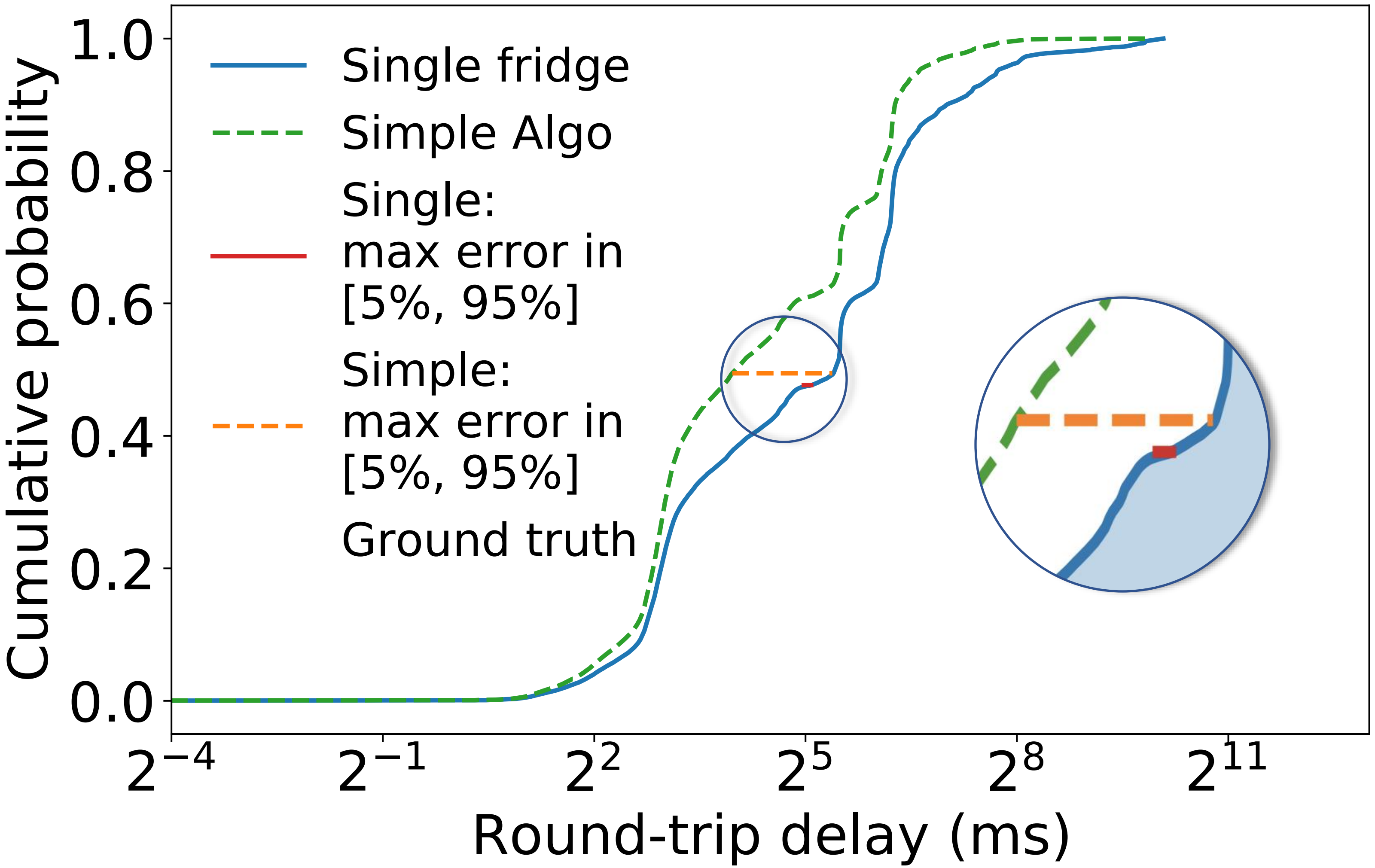
Single-fridge algo – combining reports

- $\boldsymbol{f(t)}$ = # request/response pairs in the stream with delay $\boldsymbol{t}$

  Frequency estimator $\hat{\boldsymbol{f}}(\boldsymbol{t}) = \sum$ correction factors of delay t

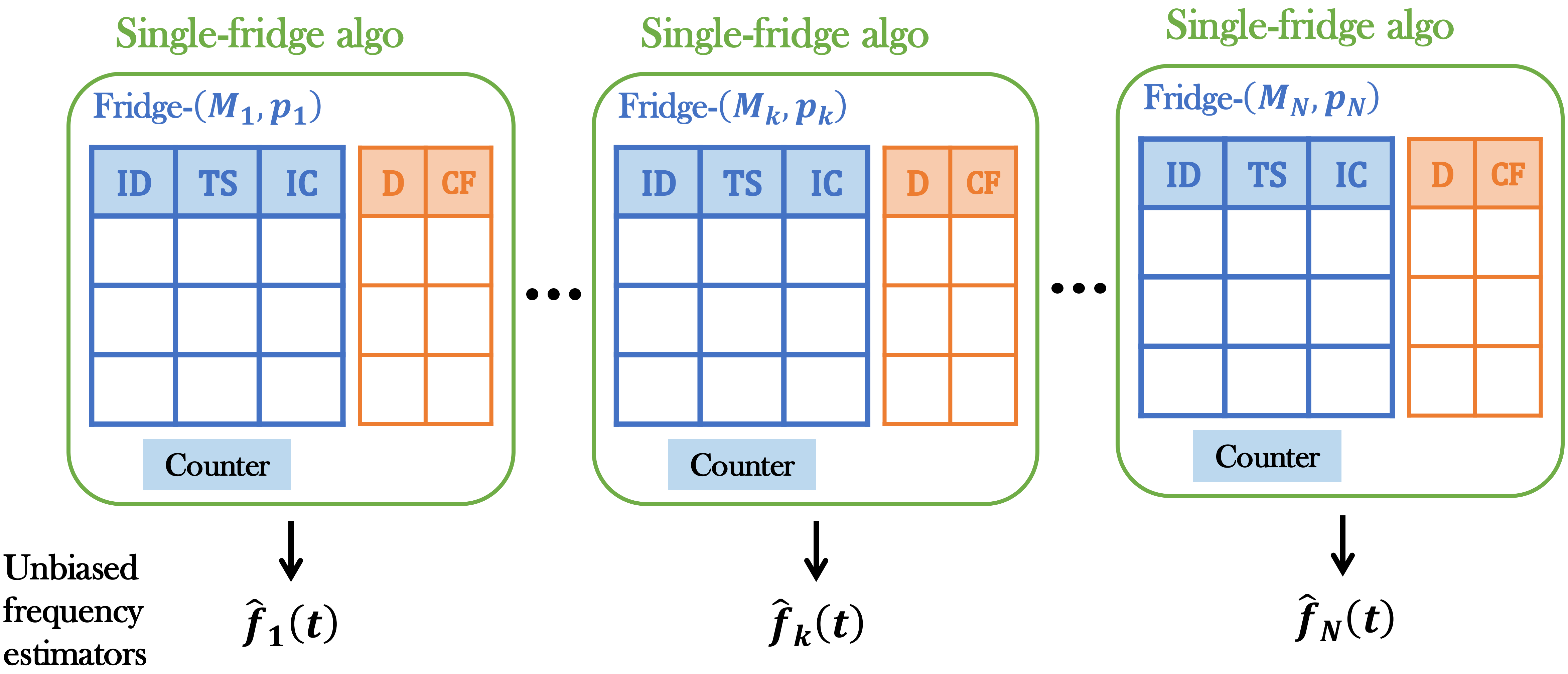
  $\hat{\boldsymbol{f}}(\boldsymbol{t})$ **unbiased**, variance known
- Obtain approximated CDF $\hat{\boldsymbol{F}}(\boldsymbol{t})$ from $\hat{\boldsymbol{f}}(\boldsymbol{t})$

Distance metric: $\left|\log_2 \frac{\textbf{Estimated delay at \%tile}}{\textbf{Ground truth delay at \%tile}}\right|$



Bi-directional campus trace, memory size $M = 2^{16}$

Multi-fridge algo



A reordered packet

SEQ number

Flow ID

Timestamp

(f, s_1, t_1) (f, s_2, t_2) (f, s_3, t_3) (f, s_4, t_4)

Set of packets of flow f $\quad S_f = \{(f, s_i, t_i)\}_{i=1}^{N_f}$

Sorted by time $\quad t_i < t_{i+1}$

i-th packet of f is **out-of-order (reordered)** if $s_i < s_{i-1}$

TCP packet reordering

TCP-induced

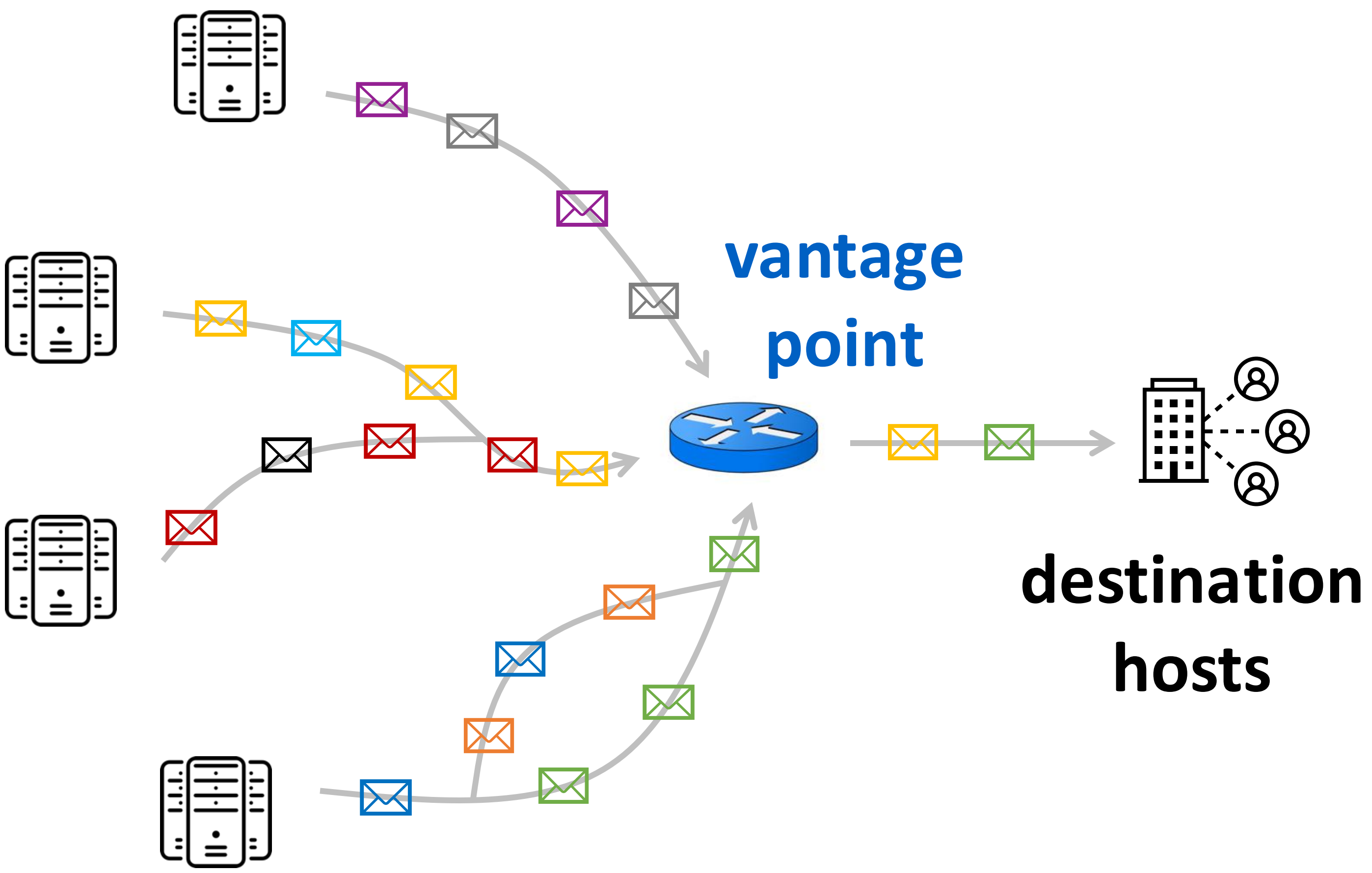
- Congestion cause TCP to lose packets and trigger retransmissions
- Symptom of a congestion problem

Network-induced

- Flaky equipment reorder packets
- TCP endpoints assumes packets loss, overreacts to perceived congestion
- Cause of a performance problem

No need to distinguish, want to detect both!

Which path is experiencing performance problems?

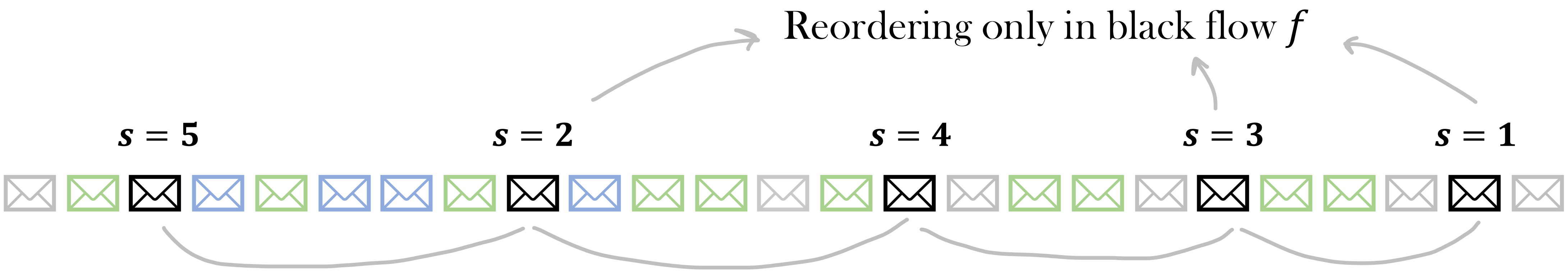


Identifying out-of-order heavy *flows*

$(\boldsymbol{f}, \boldsymbol{s} = \mathbf{3}, \boldsymbol{t})$

Flow ID	SEQ#	#reorderd	Flow size
f	$\mathbf{3} < \mathbf{3}$	1 $\mathbf{+1}$	5 $\mathbf{+1}$

Memory lower bound



f low-rate, long-lived $\Rightarrow$ Expensive to identify f

Detecting flow f needs memory **linear** in the total number of flows, even with randomness and approximation

Hard to match packets spanning a long period of time, with small memory!

Identifying out-of-order heavy *prefixes*

- Reordering is a property of a network path
- Routing decisions made at prefix level

A **prefix** g is **out-of-order heavy**
if more than ε fraction of its packets are our-of-order.

Problem statement

(1) Report **out-of-order heavy** prefixes with size at least β

(2) Avoid reporting prefixes with size at most α ($\alpha < \beta$)

Intricacies in traffic distribution

1. Memory lower bound

   ⇒ Infeasible to study all flows from a prefix and aggregate

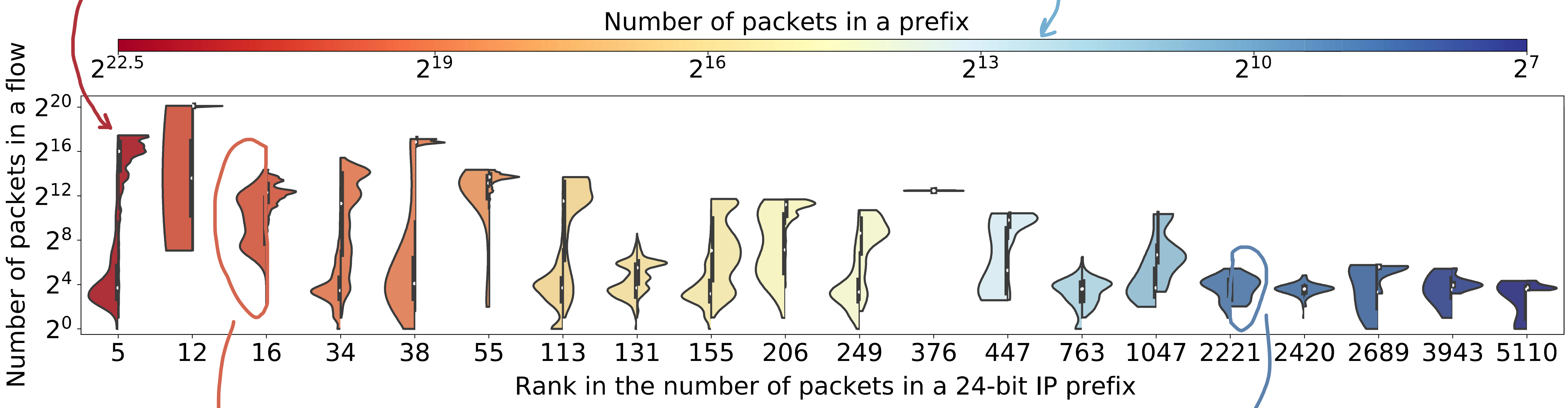
2. If each prefix of interest has at least one heavily reordered large flow

   ⇒ Existing counter-based heavy-hitter algo could help

Not the case in reality!

A violin:
a prefix of interest

Color:
prefix size



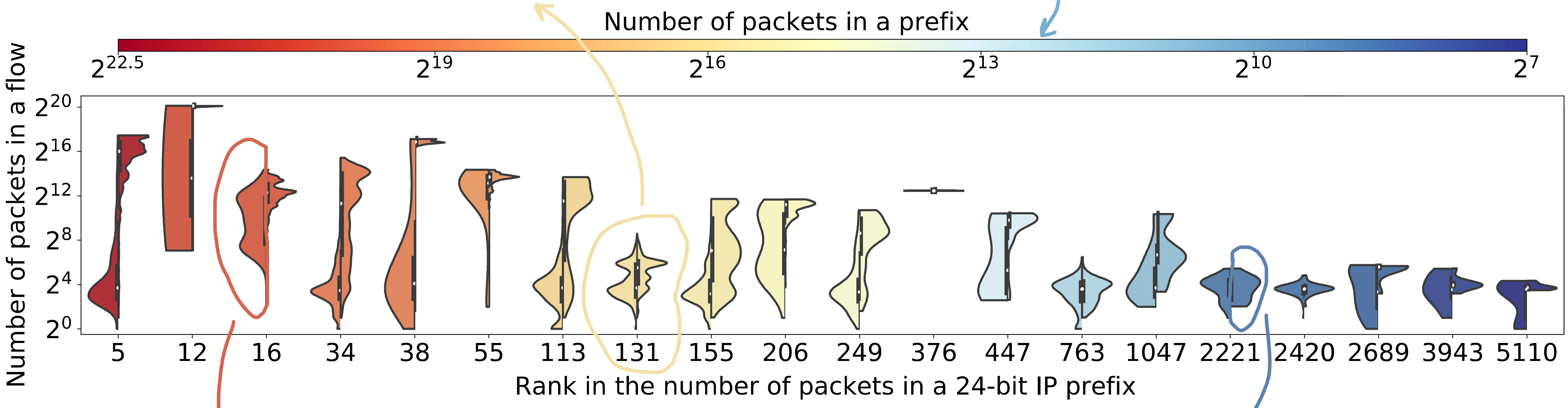
Left half:
flow size distribution in a prefix

Right half:
the fraction of reordered packets
coming from flows of what size

Largest flows in a heavily reordered prefix do not necessarily contain most of the out-of-order packets

Color: prefix size

orders-of-magnitude different



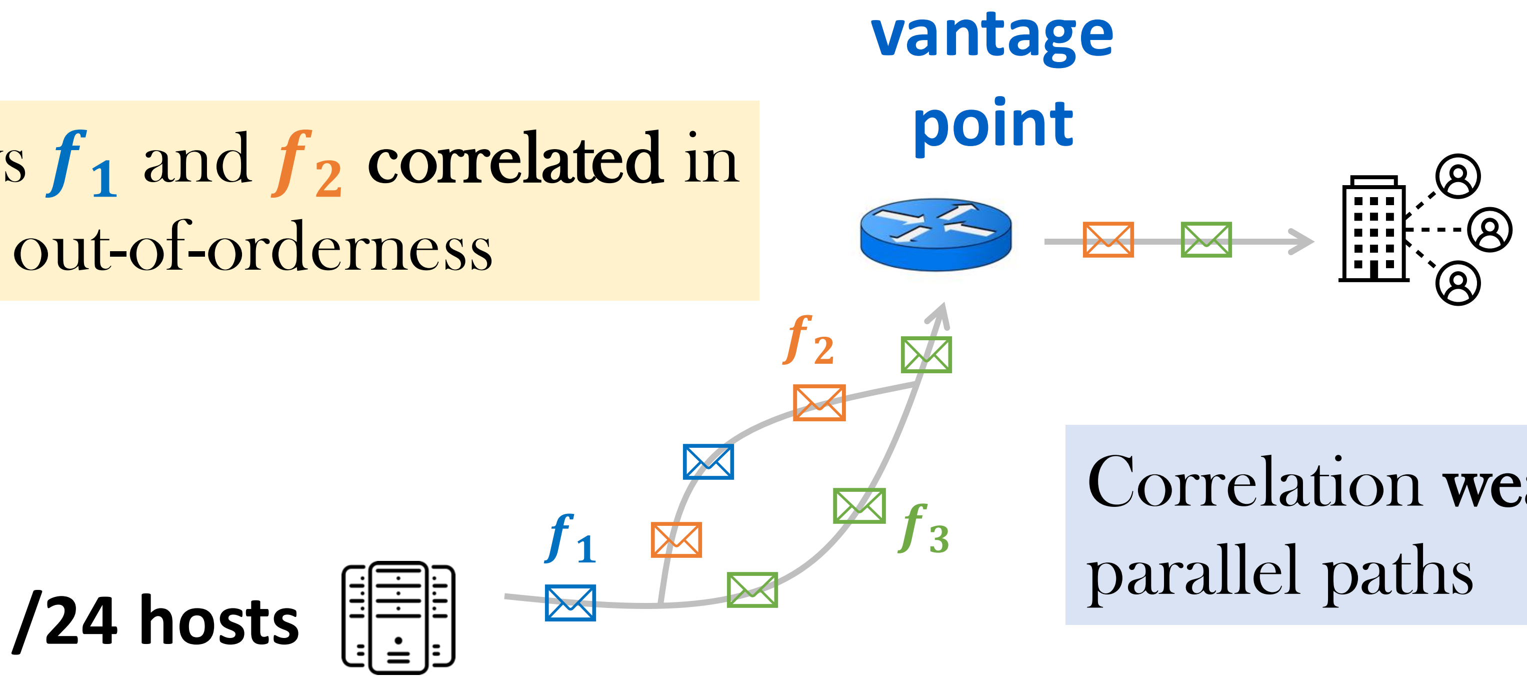
Left half:
flow size distribution in a prefix

Wide variation of flow sizes

Right half:
the fraction of reordered packets coming from flows of what size

Correlation comes to the rescue

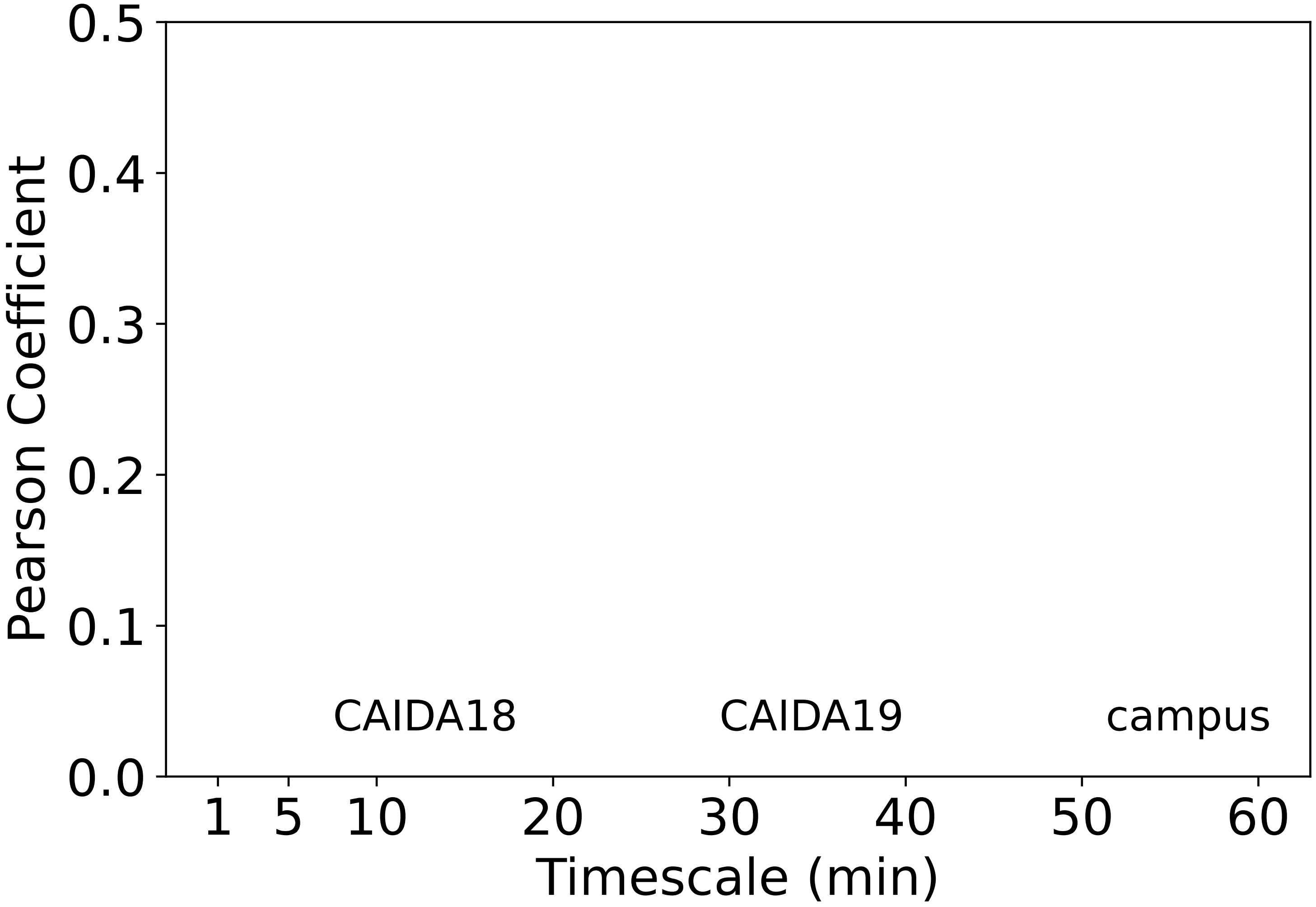
Flows f_1 and f_2 **correlated** in their out-of-orderness



Correlation **weak** due to parallel paths

The fraction of out-of-order packets in a prefix is positively correlated with that of a flow within the prefix.

A positive but weak correlation exists for all tested traces on all timescales



Indication of **weak correlation**

$\mathrm{PCC} = \mathbf{0}$ $\mathrm{PCC} \approx \mathbf{0.3}$ $\mathrm{PCC} = \mathbf{1}$

Our algorithm

Observing one flow provides no info about its prefix
⇒ observe almost all flows

Out-of-orderness of a flow is statistically identical to that of its prefix
⇒ observe one flow per prefix

Flow-sampling algorithm

Sample as many flows as possible, over a short period at a time

Hash-indexed array

Flow ID	SEQ#	#reorderd	Flow size	Timestamp

Algo: buckets

Hash-indexed array

Independent buckets

Flow ID	SEQ#	#reorderd	Flow size	Timestamp
bucket				

Algo: memory allocation

Hash-indexed array

$(\boldsymbol{f}, \boldsymbol{s}, \boldsymbol{t}) \xrightarrow{\boldsymbol{h}(\boldsymbol{g})}$

Flow ID	SEQ#	#reorderd	Flow size	Timestamp

Allocate memory at the prefix level $\Leftarrow$ Prevent prefixes with a huge number of flows from dominating the data structure

Algo: Conditional overwrite

Fix one bucket

	Flow ID	SEQ#	#reorderd	Flow size	Timestamp
$(\boldsymbol{f}, \boldsymbol{s}, \boldsymbol{t})$	$\boldsymbol{f'}$	$\boldsymbol{s'}$	$\boldsymbol{o'}$	$\boldsymbol{n'}$	$\boldsymbol{t'}$

Overwrite only if:

a) f' is **stale:** $t - t' >$ timeout T

b) The bucket has **seen many packets** from f' : $n' >$ count C

c) f' might belong to a prefix with **heavy reordering:** $o' >$ count R

Algo: Report

Fix one **bucket**

	Flow ID	SEQ#	#reorderd	Flow size	Timestamp
(f, s, t)	f'	s'	$o' > R$	n'	t'

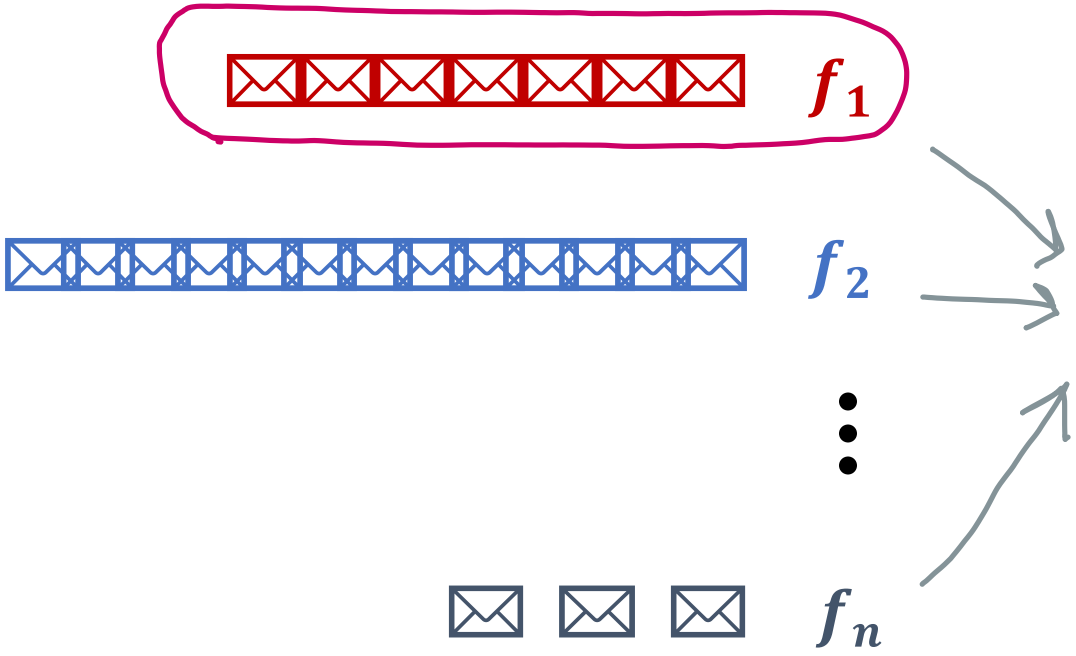
Control-plane tally

	Prefix	Prefix size	#reorderd
(g', n', o')	g'	$N' + n'$	$O' + o'$

Output g' if $N' + n' \geq \alpha$

Hash collisions are *not* so bad

Colliding with f_2 does not decrease expected #checks f_1 gets



Fix one bucket

ID	SEQ#	#OOO	FS	TS

Hybrid Scheme

Counter-based heavy-hitter data structure

fID	SEQ#	#OOO	FS

fID	SEQ#	#OOO	FS

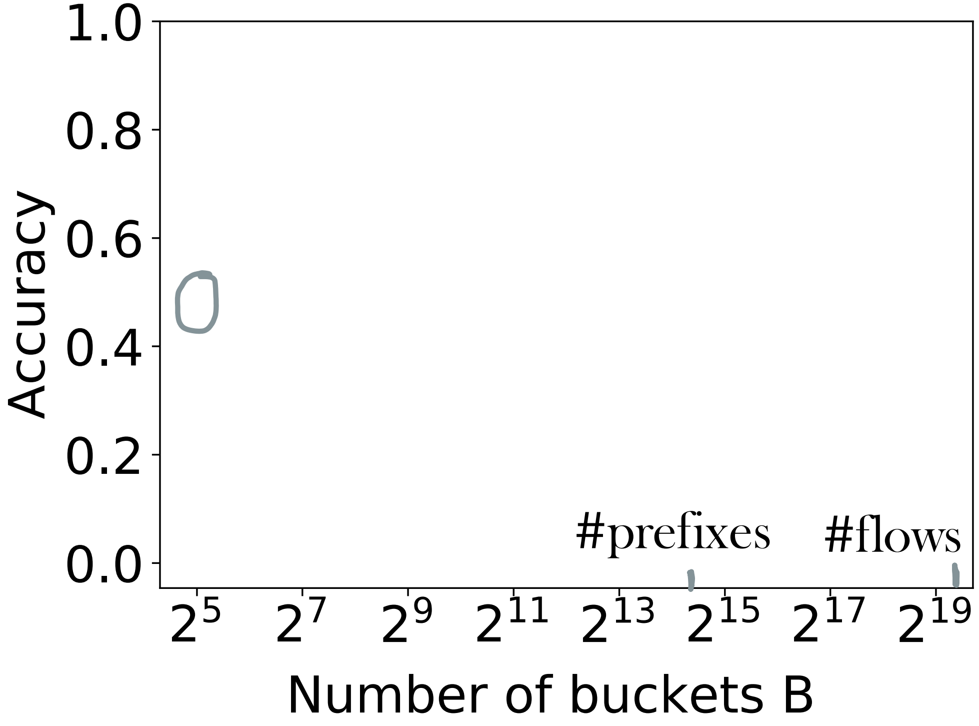
Flow sampling

fID	SEQ#	#OOO	FS	TS

Keep track of heaviness and reordering

Only admit flows not monitored in the heavy-hitter data structure

Flow-sampling HH



The flow-sampling algorithm
achieves great accuracy with small memory.

The hybrid scheme
improves the accuracy when given more memory.

To conclude the *applied* part of the talk:

- Hardware implementations available for programmable switch
- Leveraging probabilistic techniques to work with constraints
  - Measuring unbiased stats for ‘match-over-time’ queries
  - Correcting for survivorship bias
  - The use of correlation

Random Admission Policy

Initialize m empty buckets

ID	cnt
	0
	0
	0

Random Admission Policy

$\boldsymbol{x}$ →

ID	cnt
$\boldsymbol{y}$	5
$\boldsymbol{x}$	10 +1
$\boldsymbol{z}$	3

Random Admission Policy

$\boldsymbol{u}$

ID	cnt
$\boldsymbol{y}$	5
$\boldsymbol{x}$	10
$\cancel{\boldsymbol{z}}$ u	3 +1

min counter

w.p. $\frac{1}{cntVal+1} = \frac{1}{4}$

Data-plane friendly

Eventually we want to understand the performance of PRECISION

1) Random admission, as in the RAP algo
2) Approximating the global minimum to reduce the #memory accesses

Previous analysis of RAP

- Constant entering probability
  - Previous results do not transfer to the actual RAP algo
- Over i.i.d. Zipfian input streams
  - Restrictive due to time-locality

We consider arbitrarily ordered streams

Our result

Assumptions:

(1) k heavy elements, each of frequence f

(2) $\varepsilon k f$ distinct light elements, for some small $\varepsilon < 1$

Can relax these assumptions!

Given $\varepsilon < 10^{-7}$, RAP algorithm with k buckets stores at least $0.65k$ heavy elements at the end of the stream, with probability at least 0.7.

Buckets with large counter values are likely to be storing heavy elements

$\Rightarrow$ Suffices to show: constant fraction of large counters at the end

Challenges in analyzing RAP:

(1) Updating counters deterministically

(2) Overtaking counters randomly with varying probabilities

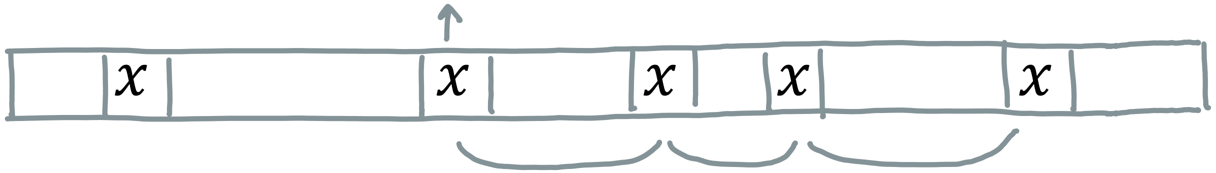
Case I: large total counter value

$\Rightarrow$ many steps where counters are incremented deterministically

$\Rightarrow$ a large fraction of counters are large

ID	cnt
x	

the t-th ***occurrence*** of x ***g***



Average gap $O(k)$

Whether the counter value increases fast is related to how bunched up the occurrences are

Case II: many steps where the ***smallest*** counter is storing a ***good*** occurrence

$\Rightarrow$ Even the **smallest** counter value would be large at EOS

Case III:

1. *Not* many steps where the smallest counter is storing a good occurrence
2. Total counter value *not* large

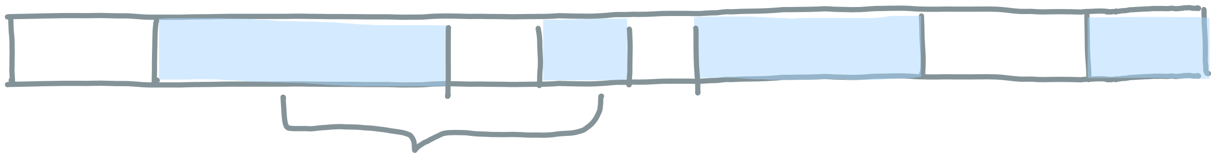
Not many steps where counters are increased deterministically

∃ some stage, #counters storing good occurrences is increased by more than $2k$ "in expectation"

$\Longrightarrow$ With some probability, contradiction

good occurrences make up a constant fraction of the stream

stream



stage j: the smallest counter value $\in \left[2^j, 2^{j+1}\right)$