Space Efficient Elephant Flow Detection

Ran Ben Basat
Technion
sran@cs.technion.ac.il

Gil Einziger
Nokia Bell Labs
srnan@cs.technion.ac.il

Roy Friedman
Technion
roy@cs.technion.ac.il

ABSTRACT
Identifying the large flows in terms of byte volume, known as elephant flows, is a fundamental capability that many network algorithms require. While optimal solutions that find the largest flows in terms of packet-count are known [5], constant update time algorithms for byte-volume were only recently discovered [1, 2]. Here, we propose an improved variant of the DIMSUM algorithm [2] that reduces the space requirement by 50% while allowing $O(1)$ update time.

CCS CONCEPTS
• Networks → Network measurement;

KEYWORDS
Network Measurement, Elephant Flows, Streaming

ACM Reference Format:

1 INTRODUCTION
Per-flow network monitoring is a fundamental building block in many networking applications such as load balancing, caching, and anomaly detection (see [3, 4] for an overview). As the number of flows is often huge, the space required for exact monitoring is too large for practical implementations. Instead, algorithms that provide approximate per-flow statistics are used.

2 PROBLEM DEFINITION
Consider a stream of packets $(x_1, w_1), \ldots$ in which each packet is associated with a flow identifier $x_i$ (e.g., 5-tuple) and a byte-size $w_i$. The byte-size of a flow $x$ is defined as $f_x = \sum_{x_i = x} w_i$ and the volume of the stream as $V = \sum w_i$. In the $\epsilon$-Elephant Flow problem, we process the stream and upon query for the size of a flow $x$ we return an estimate $\hat{f}_x$ that satisfies $f_x - \epsilon f_x \leq \hat{f}_x \leq f_x + \epsilon V$.

3 RELATED WORK
The Elephant Flow problem was introduced by [6] which presented an algorithm that uses $\epsilon^{-1}$ counters (which is known to be optimal) and has $O(\log \epsilon^{-1})$ update time. Recently, [1, 2] proposed algorithms that use $(1 + \gamma)\epsilon^{-1}$ counters, for some constant $\gamma > 0$, that has $O(1)$ amortized update time. A constant worst case time solution (DIMSUM) was proposed in [2], but it requires $(2 + \gamma)\epsilon^{-1}$ counters for some $\gamma = \Omega(1)$.

4 OUR ALGORITHM - DIMSUM++
We allocate $(1 + \gamma)\epsilon^{-1}$ counters. We break the stream into phases, maintaining the invariant that at the beginning of each phase $\gamma^{-1}/2$ counters are unallocated. Let $C_1$ denote the set of allocated counters at the beginning of an iteration and $C_2$ denote the unallocated ones. During the first $\gamma^{-1}/4$ packets of the phase, we allocate a $C_2$ counter for each arriving flow (even if he has one in $C_1$). At the same time, we find the $\gamma^{-1}/2$ smallest counters in $C_1$ (denoted $C'_1$) using a selection algorithm, while deamortizing the process that such $O(\gamma^{-1})$ operations are made at each update. While processing the next $\gamma^{-1}/4$ packets, we merge counters by summation such that at the end of the phase each flow has just one counter. Once again, we deamortize the process to get a $O(\gamma^{-1})$ update time per packet. At the end of the phase, we set $C_1 \leftarrow (C_1 \setminus C'_1) \cup C_2$ and $C_2 \leftarrow C'_1$. That is, we make sure that the largest $\epsilon^{-1}$ counters are never placed in $C'_1$ (and thus, $C_2$). Effectively, we consider the counters in $C'_1$ as deleted and ready to be reallocated in the next phase. The analysis of DIMSUM shows that as long as the flows with the $\epsilon^{-1}$ largest counters are never replaced, the procedure solves the Elephant Flow problem. Finally, picking $\gamma$ to be a small constant (e.g., 5%) we get a constant worst case time algorithm with a near-optimal space.

5 CONCLUSION
Existing solutions for the Elephant Flow problem have either: (1) non-constant update time [6]; (2) only amortized constant update time [1, 2]; or (3) require more than twice the space [2]. DIMSUM++ improves the best known $O(1)$ worst case solution by requiring half the space. Smaller memory footprint may allow the algorithm to be better cache resident in software implementations or fit into the router’s SRAM in hardware ones.

REFERENCES