2018 IEEE 26th International Conference on Network Protocols

IoT*m*: A Lightweight Framework for Fine-grained Measurements of IoT Performance Metrics

Muhammad Shahzad
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
mshahza@ncsu.edu

Anirudh Ganji
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
aganji@ncsu.edu

***Abstract*—Most Internet of Things (IoT) applications require unique guarantees on various performance metrics (such as latency, CPU availability, power fairness, *etc.*) from the IoT infrastructure. A small deterioration in these performance metrics can cause serious violations of service level agreements. To ensure that the deployed IoT infrastructure delivers the guarantees on these metrics, the first step is to measure these metrics. We present IoT*m*, a framework for measuring IoT performance metrics, which include both IoT network's quality of service (QoS) metrics and IoT node's resource utilization (RU) metrics. IoT*m* has two key properties: 1) it is lightweight and thus amenable for implementation on resource constrained IoT nodes; and 2) it can perform measurements at fine-grained levels and not just at aggregate levels. IoT*m* is comprised of two components, a lightweight IoT node unit (INU), which resides in each of the IoT nodes, and a control and query unit (CQU), which resides in a logically centralized management server. The primary role of INU is to record appropriate information about the desired performance metrics in the IoT nodes. To record the information, INU leverages a generic data structure that we propose. CQU is responsible for identifying the metrics and the IoT nodes on which those metrics should be monitored to achieve a desired measurement objective. CQU also stores the copies of data structures that the INU sends to it for long term storage. Both INU and CQU further contain query processing engines, which operate on the information stored in the data structures to answer measurement queries. To demonstrate the use of our framework, we apply it to one RU metric (number of disk accesses), and one QoS metric (round trip latency), and evaluate its accuracy. We also analyze the feasibility of its implementation on IoT nodes in terms of memory requirement and computational complexity.**

***Index Terms*—IoT; Measurements; Lightweight; Performance;**

I. Introduction

The Internet of Things (IoT) enables an exchange of data that wasn't previously easily available such as temporal and spatial distribution of moisture in soil [1], power consumption of electrical appliances in smart buildings [2], and integrity of concrete structures [3]. Each IoT application scenario requires unique guarantees on certain performance metrics (such as latency, loss, power consumption *etc.*) from the IoT infrastructure deployed in that scenario. For example, the values of vibration levels in the parts of an in-flight aircraft must be delivered to a server with as little latency as possible for real-time monitoring. The CPU utilization of a certain process on an IoT node must not exceed a certain value to ensure CPU availability for other, perhaps more critical, processes. A small deterioration in these performance metrics can cause violations of service level agreements (SLAs), and result in significant revenue and functional losses. To ensure that the deployed IoT architecture delivers the guarantees on these performance metrics and that their values lie in a desired range, the first step is to measure these metrics. IoT system and network administrators need the measurements of these performance metrics to reactively perform troubleshooting tasks such as detecting and localizing offending flows that are responsible for causing delay bursts and throughput deterioration and identifying processes that are hogging an IoT node's CPU. They also need these measurements to proactively locate and update any potential future bottlenecks.

In this paper, our objective is to take the first step towards developing a framework for measuring IoT performance metrics, which include both IoT network's Quality of Service (QoS) metrics such as latency, loss, and throughput and IoT nodes' Resource Utilization (RU) metrics such as power consumption, disk accesses and utilization, radio-on time *etc.* This framework must have two key properties. First, it should be *lightweight* for IoT nodes, *i.e.*, it should require very small amount of memory and compute resources on IoT nodes when performing measurements, and should keep all the complexity on the cloud side. This is important because most IoT nodes are resource constrained in terms of either the available power, or the compute and memory resources, or even both. Second, it should be able to perform *fine-grained* measurements, *i.e.*, it should not only be able to obtain the aggregate measurements of the desired performance metric (such as the aggregate utilization of CPU by *all* processes on an IoT node or the average latency of *all* the packets going from an IoT node to a cloud platform), but it should also be able to obtain per-instance measurements of that metric (such as the CPU utilization of *each* process running on the IoT node, and the average latency of *each* flow going from the IoT node to the cloud platform, respectively). In the rest of this paper, we will use the term instance to represent an individual entity for which the desired performance metric exists. Examples of instance include a network flow or a CPU process. The desired performance metrics for these instances can be flow size and CPU utilization, respectively. Fine-grained measurements are important because even if the aggregate value of a performance

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00012

IEEE computer society

metric appears normal, its value for a particular instance may be wildly abnormal. For example, IoT gateways can experience microbursts (simultaneous short bursts of data from a large subset of IoT nodes) that cause packet losses for some flows, even when the average traffic at that gateway across all flows over a period of time is well within the limits of the gateway capacity. While solutions to such problems have been proposed in the past for conventional networks and for data centers, and it may even be possible to adapt those solutions for IoT architectures, it is still imperative to have a measurement scheme in place to rapidly detect and localize such problems.

While measuring performance metrics in IoT is a new and largely an unexplored area, a significant amount of work has been done on measuring performance metrics in conventional networks and data centers, which reiterates the importance of measuring network performance metrics in the emerging IoT networks. Although several schemes have been proposed to measure performance metrics in conventional networks and systems, they are not suitable for IoT architectures for two main reasons. First, the majority of the existing schemes are not passive, *i.e.*, they perform active operations, such as using active probes, to obtain the measurements [4]–[11]. While active schemes work well with conventional network devices, they are problematic for IoT nodes because they interfere with the regular operations of the IoT nodes, and the limited amount of resources on such nodes makes it hard for the nodes to perform such auxiliary active operations while ensuring that their regular operations proceed unaffected. Second, the existing passive measurement schemes for conventional networks and systems require a significant amount of computational and memory resources. For example, Moshref *et al.* leveraged the abundance of computational resources in the servers in data centers to develop a measurement framework to detect events of interest [12]. While computational resources are usually not a problem on conventional servers, they are not as abundant on IoT nodes. Consequently, it is not possible for IoT nodes to implement conventional measurement methods without impeding their regular operations. Thus, there is a need to develop a framework that is designed keeping IoT architectures in mind.

In this paper, we present IoT*m*, a lightweight framework for fine-grained measurement of IoT performance metrics, which include both QoS as well as RU metrics. It is comprised of two components, an IoT node unit (INU), which resides in each IoT node, and a control and query unit (CQU), which resides in a logically centralized management server. The management server itself can either be deployed locally or in the cloud.

INU is comprised of two sub-components: 1) a data structure in which the IoT node stores appropriate information about the desired performance metric; and 2) a local query processing engine, which receives queries from CQU (we will discus CQU shortly) and answers them using the information stored in the data structure. The INU sends the data structure to the CQU either periodically or when the data structure gets full and the CQU stores it for analysis and long-term storage. The data structure has three properties that make it ideal for implementation in IoT nodes. First, it enables both fine-grained and aggregate measurements of a variety of performance metrics, including, but not limited to, latency, disk accesses, CPU utilization, and several others. Second, it is computationally very lightweight and requires only a single hash computation and no more than two memory updates per insertion. Third, it has a very small memory footprint.

CQU is comprised of three sub-components: 1) a control unit, which, based on the high level measurement objective, is responsible for selecting the IoT nodes on which the measurements should be taken, the performance metrics that should be measured on those IoT nodes, and the granularity at which those performance metrics should be measured; 2) a storage, where CQU stores the data structures sent by the INUs; and 3) a global query processing engine, which answers user's queries from the stored data structures. Next, we give an example to demonstrate the use of these three sub-components.

Consider a scenario where an IoT infrastructure provider implements a large number of traffic monitoring sensors that measure and provide the extent of congestion on all roads and intersections throughout the city. Suppose an IoT service controls the timings for traffic lights at all intersections in that city and uses an algorithm to calculate the optimal durations for the red, yellow, and green lights based on the state of congestion on different roads and intersections in the city. Lets assume that the algorithm that this IoT service uses requires that the latest state of congestion at city intersections be delivered to it within 100ms, while the state of congestion at portions of roads farther away from the intersections can take longer to be delivered. This requirement gives rise to the need for measuring latencies of packets sent by traffic sensors at city intersection to ensure that they never experience delay $>$100ms . With this high level latency measurement objective in view, the control unit instructs the INUs in the traffic sensors at the intersections (and not at the portions of roads away from the intersections) to start recording information in their data structures that can later be used for measuring latencies. The query processing engine in CQU can later be used to estimate and analyze latencies experienced by packets at any desired time from the data structures sent by INUs to CQU and stored in CQU's storage to ensure that the latencies experienced by these packets were never over 100ms, and if they were, appropriate actions can be taken to keep them under 100ms.

From the discussion above, we see that our measurement framework has 5 sub-components: the data structure and the local query processing engine in INU, and the control unit, storage, and global query processing engine in CQU. In this paper, we focus on the design of 3 of these 5 sub-components: the data structure and the local query processing engine in INU, and the global query processing engine in CQU. For control unit, we assume that a network administrator configures it manually based on the high level measurement objectives. Automating the operations of control unit will be the part of our future work. For the storage, we assume that an appropriate database is in place that can store the data structures sent by INUs to the CQU.

Key Contributions: In this paper, we make three key contributions: 1) we present, IoT*m*, a lightweight framework for fine-grained measurement of IoT performance metrics, which include both QoS and RU metrics; 2), we present a generic data structure that enables INUs to store information about IoT performance metrics in compute and memory efficient way; 3) we demonstrate the application of our framework using two arbitrarily chosen IoT performance metrics (disk accesses and round trip latencies) as examples, and present the accuracy of our framework through extensive experimental evaluations.

II. Data Structure

In this section, we present our data structure that can efficiently store measurements for a variety of different performance metrics. We first describe how the INU on any given IoT node inserts measurements into this data structure. After that, we discuss how INU sends it to CQU for long term storage and analysis. Last, we present its complexity analysis and theoretical modeling. To describe how the INU inserts measurements into this data structures, we use an arbitrary metric $\mathfrak{M}$ that has to be measured for one or more distinct instances. Let we use I to represent an arbitrary instance. As an example, this data structure can be used to store the packet count (*i.e.*, the metric $\mathfrak{M}$) of each network flow (*i.e.*, the instance I) that the IoT node generates. As another example, this data structure can also be used to store the number of disk accesses (*i.e.*, the metric $\mathfrak{M}$) of each process (*i.e.*, the instance I) running in a given IoT node.

A. Construction

Our data structure is comprised of an array $\mathbf{B}$ of n buckets, where each bucket $\mathbf{B}[i]$ $(1 \leq i \leq n)$ has b bits with initial value 0. The data structure requires instances to have unique IDs. An ID can be any information that distinguishes one instance from the others. For example, if the instance is a network flow, its ID can be the standard five tuple (*i.e.*, source IP, destination IP, source port, destination port, and protocol type). Similarly, if the instance is a process running in the IoT node, its ID can be the unique process ID assigned to it by the OS. For any arbitrary instance with ID I, our data structure maps it to a bucket subarray, which is a unique subset of the buckets in the bucket array. Each bucket subarray comprises m buckets, where $m << n$. To make the mapping unique and memoryless (*i.e.*, without requiring any lookup or hash tables), our data structure maps each instance to a random subarray such that the probability of two different instances being mapped to the same subarray is practically negligible. A bucket may belong to multiple bucket subarrays. Figure 1 shows an example bucket array and its four bucket subarrays for four instances I_1 through I_4. We observe from this figure that buckets 5 and 7 belong to multiple bucket subarrays.

To insert each measurement of the desired metric $\mathfrak{M}$ of an instance I, the INU first randomly selects one bucket in the bucket subarray corresponding to the instance I, and then adds the measured value of the metric in that bucket. Formally, to add a measurement for instance I, INU chooses

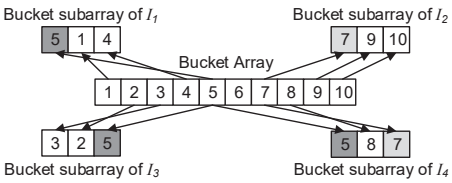


Fig. 1. Illustration of bucket array and bucket subarrays

a random number j from a uniform distribution in the range $[1, m]$, calculates the hash function $H(I, j)$ whose output is uniformly distributed in the range $[1, n]$, and increments the bucket $\mathbf{B}[H(I, j)]$ by the value of the measurement. Thus, the sum of all the measurements of the metric $\mathfrak{M}$ of instance I will be uniformly distributed across m buckets: $\mathbf{B}[H(I, 1)], \mathbf{B}[H(I, 2)], \cdots, \mathbf{B}[H(I, m)]$. These m buckets constitute the bucket subarray of the instance I, which is denoted by $\mathbf{B}^I$ where $\mathbf{B}^I[j] = \mathbf{B}[H(I, j)]$ for $j \in [1, m]$.

Note that while storing the sum of all the measurements is appropriate in the case of some metrics, such as the total number of memory accesses, it is not appropriate for some other metrics, such as CPU utilization. The appropriate measures for such metrics are the average value of those metrics. To calculate the average value of such a metric for any given instance I, in addition to the sum of all the measurements for I, the query processing engine will also need to know the number of times the measurements for I were inserted to I's bucket subarray. For this, the INU maintains another array $\mathbf{C}$ containing the same number of buckets as the array $\mathbf{B}$ (*i.e.*, n) with initial values of 0, and the same number of buckets per subarray (*i.e.*, m). Every time INU adds a measurement to some bucket $\mathbf{B}[i]$, it also increments the corresponding bucket $\mathbf{C}[i]$ by 1. This way, the bucket array $\mathbf{B}$ stores the sum of the measurements of the desired metric for each instance and the bucket array $\mathbf{C}$ stores the number of times the measurements of the desired metric for each instance were inserted in the bucket array $\mathbf{B}$. In Section III, we will describe how a query processing engine applies appropriate statistical operations to extract the average and/or the sum of the measurements of the metric for any given instance I from the bucket array.

B. Management

The INU on each IoT node sends the data structure, *i.e.*, the bucket array $\mathbf{B}$ (and the bucket array $\mathbf{C}$, if being used) to the CQU for storage either when any of the bucket becomes full or after a certain amount of time has passed since the bucket array $\mathbf{B}$ was initialized to 0. We call the bucket array $\mathbf{B}$ that is sent to the CQU a *bucket epoch* $\mathbf{B}$. For each bucket epoch, INU also sends time stamps of the first and the last recorded measurement, which the CQU uses to distinguish between several bucket epochs sent by the INU on any given node. After sending the data structure, INU resets all buckets to zero and starts recording measurements again.

Note that instead of using the data structure that we have proposed, INU may be able to store the measurements of the desired metric for each instance by simply maintaining a separate counter for each instance. When the number of

instances is small, this approach may even require less memory. However, this approach has 3 limitations. First, it does not scale: as the number of instances increase, the amount of lookup state that the IoT node needs to maintain increases to keep track of which counters are associated with which instances. This may not always be possible for resource constrained IoT nodes. In comparison, the data structure in our IoT*m* framework does not require the INU to maintain any lookup state, rather maps the instances to appropriate counters using our memoryless hashing approach. Second, as the number of instances increase, separate counters for each instance consume a lot more memory compared to our data structure, as we will show in Section IV. Third, it is often not possible to anticipate the number of measurements of any given instance in advance. Thus, allocating the same number of bits to counters for each instance can result in overflow in some counters and under utilization in others. In comparison, in our data structure, each bucket is shared among many instances and each instance is mapped to many buckets, which mitigates the problems of under utilization and overflow, respectively.

C. Analysis

1) Complexity: When inserting information about a measurement in bucket array, the IoT node has to perform only one hash computation (to identify the bucket) and up to only two memory updates (one to add the value of the measurement to the bucket array **B** and another to keep the count of the number of measurements in bucket array **C**, if used). Due to such minimal computation and memory access requirements, our data structure is very lightweight and amenable for implementation in IoT nodes. We will study the memory footprint of our data structure in Section IV when we use the IoT*m* framework to measure various IoT performance metrics.

2) Modeling: Next, we derive the expression for the probability distribution of the values of the buckets in the bucket arrays, which the query processing engine will use to estimate the average and/or the sum of the measurements of the metric stored in the bucket arrays for any given instance. Let B_I be the random variable representing the amount contributed by instance I to a bucket $\mathbf{B}^I[j]$ $(1 \leq j \leq m)$ in its bucket subarray. Let s_I be the sum of all the measurements of instance I that contributed to the current bucket epoch **B**. As each bucket in the bucket subarray of I has a probability $\frac{1}{m}$ of being selected for insertion of any measurement for the instance I, B_I follows a binomial distribution, *i.e.*, $B_I \sim \text{Binom}(s_I, 1/m)$. Thus, the amount contributed by an instance to each bucket in its bucket subarray follows a binomial distribution.

The amount contributed by all instances other than the instance I to each bucket in the bucket subarray of instance I also follows a binomial distribution. Let B_r be the random variable representing the amount contributed by measurements of all instances other than the instance I to bucket $\mathbf{B}^I[j]$. The probability that a measurement of an instance $\bar{I} \neq I$ contributes an amount to bucket $\mathbf{B}^I[j]$ is the product of the probability that the hash function H maps the measurement to $\mathbf{B}^I[j]$ given that $\mathbf{B}^I[j] \in \mathbf{B}^{\bar{I}}$, which is $1/m$, and the probability that bucket $\mathbf{B}^I[j]$ is in the bucket subarray of $\bar{I}$, which is denoted by $P\{\mathbf{B}^I[j] \in \mathbf{B}^{\bar{I}}\}$ and calculated as follows:

$$P\{\mathbf{B}^I[j] \in \mathbf{B}^{\bar{I}}\} = 1 - \left\{ \binom{m}{0} \left(\frac{1}{n}\right)^0 \left(1 - \frac{1}{n}\right)^m \right\}$$
$$= 1 - \left\{ 1 - \frac{m}{n} + \frac{(m)(m-1)}{n^2 \times 2!} - \dots \right\} \approx \frac{m}{n} \quad (1)$$

Thus, the probability that a measurement for an instance $\bar{I} \neq I$ contributes an amount to bucket $\mathbf{B}^I[j]$ is $\frac{1}{m} \times \frac{m}{n} = \frac{1}{n}$. Representing the sum of all the amounts contributed by the measurements of all instances in the given bucket epoch by s, the binomial distribution of B_r is $B_r \sim \text{Binom}(s - s_I, 1/n)$.

Next, we calculate the probability distribution of any given bucket in the bucket array. As $B = B_I + B_r$, and B_I and B_r are almost independent when $s_I << s$, the probability distribution function of B is calculated as follows.

$$P\{B = b\} \approx \sum_{b_I=0}^{b} \left\{ P\{B_I = b_I\} \times P\{B_r = b - b_I\} \right\}$$

Note that in practice, s_I is indeed much smaller than s because s_I is the amount added by a single instance in the bucket epoch while s is the amount added by all the instances. Replacing $P\{B_I = b_I\}$ and $P\{B_r = b - b_I\}$ with their respective binomial distribution expressions, we get

$$P\{B = b\} \approx \sum_{b_I=0}^{b} \left\{ \binom{s_I}{b_I} \left(\frac{1}{m}\right)^{b_I} \left(1 - \frac{1}{m}\right)^{s_I - b_I} \times \binom{s - s_I}{b - b_I} \left(\frac{1}{n}\right)^{b - b_I} \left(1 - \frac{1}{n}\right)^{s - s_I - b + b_I} \right\} \quad (2)$$

Following the same steps, we can obtain the expression for the probability distribution of the values of the buckets in the bucket array **C**. Let t_I represent the number of times the measurements of instance I were inserted in **B**, and t represent the total number of insertions into the bucket array **B**. Let C be the random variable representing the value of any arbitrary bucket $\mathbf{C}^I[j]$ $(1 \leq j \leq m)$ in the bucket subarray. The expression for the probability distribution of the values of the buckets in the bucket array **C** turns out to be the following.

$$P\{C = c\} \approx \sum_{c_I=0}^{c} \left\{ \binom{t_I}{c_I} \left(\frac{1}{m}\right)^{c_I} \left(1 - \frac{1}{m}\right)^{t_I - c_I} \times \binom{t - t_I}{c - c_I} \left(\frac{1}{n}\right)^{c - c_I} \left(1 - \frac{1}{n}\right)^{t - t_I - c + c_I} \right\} \quad (3)$$

III. Query Processing Engine

The control unit issues queries to the query processing engines. The query can be provided manually by the administrator through the control unit or the control unit can issue them automatically to measure these performance metrics to ensure that the IoT system is delivering on any required SLAs. A query comprises three attributes: 1) the instance ID, 2) the starting and ending times of the period during which the value of the desired metric is required, 3) whether the response to

the query should be the sum or the average of the values of the desired metric. Based on the starting and ending times, the control unit first determines whether the bucket epochs whose time frames overlap with these starting and ending times are all stored in the storage of CQU or the time frame also covers the data structure currently under construction by INU in an IoT node. Next, the control unit instructs the appropriate query processing engines (*i.e.*, either in CQU, or INU, or both) to estimate the sum of the value of the desired metric from each identified bucket epoch **B**, and send them back to the control unit. Control unit adds these values estimated from the bucket epochs to obtain the overall sum for the desired instance. If the average value of the metric is desired, the query processing engine further extracts the number of times the measurement of the desired metric is inserted into bucket epochs **B** from all corresponding bucket epochs **C**, and sends them to the control unit. The control unit adds them to get an estimate of the total number of times the measurements of the desired metric were inserted into all bucket epochs and divides the sum from all bucket epochs with it to get the average value of the metric for the desired instance.

Next, we describe how the query processing engine extracts the value of the sum from any given bucket epoch **B**. The method to extract the value of the number of times a measurement is inserted from any given bucket epoch **C** is exactly the same. Furthermore, both local and global query processing engines use the exact same method to extract the values of the sum and the number of times the measurements are inserted from bucket epochs **B** and **C**, respectively. Our objective here is to estimate the value of s_I for any given instance with ID I from any given bucket epoch **B**. Let $\tilde{\mathbf{B}}^I[j]$ denote the observed value of bucket $\mathbf{B}^I[j]$ and let $\tilde{s}_I$ denote the estimate that the query processing engine returns for the value of s_I. The probability or *likelihood* of getting the observed value $\tilde{\mathbf{B}}^I[j]$ of a bucket $\mathbf{B}^I[j]$ in the bucket subarray of instance I is given by Eq. (2). Thus, the likelihood of getting the observed values of all buckets in the bucket subarray of the flow is given by the following equation.

$$\begin{aligned} L &= \prod_{j=1}^{m} P\left\{B = \tilde{\mathbf{B}}^I[j]\right\} \\ &= \prod_{j=1}^{m} \left\{ \sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \left[\binom{s_I}{b_I} \left(\frac{1}{m}\right)^{b_I} \left(1-\frac{1}{m}\right)^{s_I-b_I} \times \right. \right. \\ &\left. \left. \binom{s-s_I}{\tilde{\mathbf{B}}^I[j]-b_I} \left(\frac{1}{n}\right)^{\tilde{\mathbf{B}}^I[j]-b_I} \left(1-\frac{1}{n}\right)^{s-s_I-\tilde{\mathbf{B}}^I[j]+b_I} \right] \right\} \end{aligned} \quad (4)$$

Note that the right hand side of the equation above has only one unknown, *i.e.*, s_I. We use the maximum likelihood estimation method to estimate the value of s_I using this equation. Formally, the estimated value $\tilde{s}_I$ of the sum of measurements of the instance I is given by $\tilde{s}_I = \arg\max_{s_I}\{L\}$. Taking natural log of L, differentiating $\ln\{L\}$ with respect to s_I, and equating $\frac{d}{ds_I}\ln\{L\}$ to 0, we get

$$\begin{aligned} &\sum_{j=1}^{m} \frac{d}{ds_I} \ln \left\{ \sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \left[\binom{s_I}{b_I} \left(\frac{1}{m}\right)^{b_I} \left(1-\frac{1}{m}\right)^{s_I-b_I} \times \right. \right. \\ &\left. \left. \binom{s-s_I}{\tilde{\mathbf{B}}^I[j]-b_I} \left(\frac{1}{n}\right)^{\tilde{\mathbf{B}}^I[j]-b_I} \left(1-\frac{1}{n}\right)^{s-s_I-\tilde{\mathbf{B}}^I[j]+b_I} \right] \right\} = 0 \end{aligned} \quad (5)$$

Let

$$\mathbb{X}[j] = \left(\frac{1}{m}\right)^{b_I} \left(1-\frac{1}{m}\right)^{-b_I} \left(\frac{1}{n}\right)^{\tilde{\mathbf{B}}^I[j]-b_I} \left(1-\frac{1}{n}\right)^{s-\tilde{\mathbf{B}}^I[j]+b_I}$$

$$\mathbb{Y}[j] = \binom{s_I}{b_I} \binom{s-s_I}{\tilde{\mathbf{B}}^I[j]-b_I} \left(1-\frac{1}{m}\right)^{s_I} \left(1-\frac{1}{n}\right)^{-s_I}$$

Thus, Eq. (5) becomes

$$\sum_{j=1}^{m} \frac{d}{ds_I} \ln \left\{ \sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} [\mathbb{X}[j] \times \mathbb{Y}[j]] \right\} = 0$$

As $\mathbb{X}[j]$ is not a function of s_I, the equation above becomes

$$\sum_{j=1}^{m} \left\{ \frac{\sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \left\{ \mathbb{X}[j] \times \frac{d}{ds_I}\mathbb{Y}[j] \right\}}{\sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \{\mathbb{X}[j] \times \mathbb{Y}[j]\}} \right\} = 0 \quad (6)$$

To calculate $\frac{d}{ds_I}\mathbb{Y}[j]$, we use the following identity.

$$\frac{d}{dw}\binom{v}{w} = \binom{v}{w}\left(\psi^{(0)}\{v-w+1\} - \psi^{(0)}\{w+1\}\right)$$

Using this identity and standard algebraic operations, we get the following equation for $\frac{d}{ds_I}\mathbb{Y}[j]$. Let $\mathbb{Z}[j] = \frac{d}{ds_I}\mathbb{Y}[j]$. Thus,

$$\begin{aligned} \mathbb{Z}[j] = \mathbb{Y}[j] \Bigg[&\log\left\{1-\frac{1}{m}\right\} - \log\left\{1-\frac{1}{n}\right\} \\ &-\psi^{(0)}\{1+s-s_I\} + \psi^{(0)}\left\{1+b_I-\tilde{\mathbf{B}}^I[j]+s-s_I\right\} \\ &+\psi^{(0)}\{1+s_I\} - \psi^{(0)}\{1-b_I+s_I\} \Bigg] \end{aligned}$$

Thus, the estimated value $\tilde{s}_I$ of s_I, *i.e.*, the sum of measurements of instance I, in the given bucket epoch **B** is given by the numerical solution of the following equation:

$$\sum_{j=1}^{m} \left\{ \frac{\sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \{\mathbb{X}[j] \times \mathbb{Z}[j]\}}{\sum_{b_I=0}^{\tilde{\mathbf{B}}^I[j]} \{\mathbb{X}[j] \times \mathbb{Y}[j]\}} \right\} = 0 \quad (7)$$

The estimated value $\tilde{t}_I$ of t_I, *i.e.*, the number of times the measurements are inserted in the given bucket epoch **B**, is also given by the numerical solution of Eq. (7) after replacing $\tilde{\mathbf{B}}^I[j]$, b_I, s, and s_I with $\tilde{\mathbf{C}}^I[j]$, c_I, t, and t_I, respectively.

Discussion: As described in Section II-A, the INU spreads the measurements of any instance into m buckets. By distributing the measurements into multiple buckets and by keeping $m << n$, INU significantly reduces the dependence of the accuracy of estimates on the distribution of the measurements. This is because when $m << n$, no two flows will have a large number of common buckets in their bucket

subarrays. Thus, even if one instance with large value shares a bucket with another instance with small value, the remaining $m-1$ buckets of the instance with small value will not be shared with the same large instance. Consequently, the net estimate from the m buckets does not have a large error. Our proposed framework finds applications in the majority of IoT deployment scenarios. For example, it can be deployed to keep track of the amount of time each node transmits for subsequent analysis of the fairness in transmission workload of low-power nodes. As another example, it can be used to monitor the latency experienced by the packets of individual IoT nodes, which is a critical metric in time-sensitive IoT deployments such as in autonomous machines on factory floors.

IV. Applications and Evaluation

In this section, we demonstrate the use of our IoT*m* framework by applying it to store and estimate one RU metric, namely the number of disk input/output (IO) operations and one QoS metric, namely round trip latency. Note that our framework is not limited to these two metrics, and can be used to store and estimate several other IoT metrics, such as CPU utilization, throughput, packet loss, memory consumption, *etc.* For the number of disk IO operations, the appropriate measurement is the *total* number of IO operations per process, while for latency, the appropriate measurement is the *average* value per flow. Thus, the two IoT performance metrics that we have chosen cover both types of metrics: one that does not require the use of the supplementary bucket array **C** and one that does. In the rest of this section, for each of the two metrics, we first describe how the INU inserts the measurements of that metric into the data structure. After that we describe the real world traces that we used to evaluate the performance of our framework for that metric. Last, we present the results on the accuracy of the estimates of that metric provided by the query processing engine, and on the physical memory required on the IoT nodes to maintain the data structure.

A. Disk IO Operations per Process

1) Method: When the control unit instructs INU on any IoT node to start recording disk IOs of processes, the INU initializes the bucket array **B** comprising n buckets. As we are interested in the *total* number of disk IOs per process and not the *average*, the INU does not initialize the bucket array **C**. Based on the requirements, the control unit can even specify exactly for which processes the disk IOs should be recorded.

Every time a process on the given IoT node performs a disk IO operation, the INU on that node selects a random number j from a uniform distribution in the range $[1, m]$, appends it with the ID p of the process, calculates the hash function $H(p, j)$ whose output is uniformly distributed in the range $[1, n]$, and increments the bucket $\mathbf{B}[H(p, j)]$ by one. To estimate the number of disk IOs of any given process with ID p over a desired period of time, the control unit first identifies all bucket epochs whose time frames overlap with that desired period of time. Depending on where the identified bucket epochs are currently stored, the control unit asks the query processing engine either in CQU, or in INU, or in both, to estimate the number of disk IOs from their corresponding bucket epochs. The query processing engine(s) use Eq. (7) to estimate the number of disk IOs for that process from the identified bucket epochs, and send the estimates back to the control unit. The control unit adds the estimate from each bucket epoch to get the final estimate of the total number of disk IOs of the process p over the desired period of time.

2) Traces: To evaluate the accuracy of our framework in estimating the number of disk IOs per process, we collected disk IO traces from a Raspberry Pi executing MQTT protocol (the commonly used application layer protocol for IoT [13]) and sending/receiving packets from an MQTT broker. Our trace file contains log entries, where each log entry corresponds to a disk IO operation and comprises the time stamp of that disk IO operation as well as the ID of the process that performed that disk IO. We captured disk IOs for 10 processes for a duration of 10 minutes.

3) Evaluation: Next, we first study the accuracy of the estimates using fixed values of the two parameters n (*i.e.*, the number of buckets in the bucket array) and m (*i.e.*, the number of buckets in each bucket subarray). After that, we study the effect of the change in the values of these two parameters on the accuracy of our framework. After that, we study the effect of the number of bucket epochs across which queries are spread, *i.e.*, the number of bucket epochs from which the query processing engine(s) have to estimate values in order for the control unit to be able to answer the query. Last, we discuss the memory requirements of our data structure in an IoT node. In all our experiments, we used b (*i.e.*, the size of each bucket) as 16 bits.

To study the accuracy of IoT*m*, we implemented the INU emulator in Matlab that traverses the log file containing the traces and inserts the disk IO counts to the data structure using the desired values of n, m, and b. More specifically, the INU emulator simulates the time duration of 10 minutes. Every time the simulator time matches the time stamp of a log entry, the INU emulator increments a bucket in the bucket subarray corresponding to the process ID in that log entry, as described in Section IV-A1. The INU transfers the data structure to CQU (also implemented in Matlab) every t minutes. This way, from the simulation with t minute wide bucket epochs, the CQU emulator receives $10/t$ bucket epochs. In different simulations, we used one of the following three values for t: $\{1, 5, 10\}$.

The motivation behind performing simulations with different durations of bucket epochs (*i.e.*, using different values of t in different simulations) is twofold. First, it enables us to evaluate the accuracy of IoT*m* for queries that span a single bucket epoch as well as those that span multiple bucket epochs. Second, in evaluating the accuracy, the query processing engine is exposed to several ranges of the number of disk IOs per process. Figure 2 shows boxplot of the number of disk IOs of the 10 processes stored in each bucket epoch from three simulations corresponding to three values of t, *i.e.*, $t = 1, 5$, and 10. For this figure, $n = 20$ and $m = 5$. Each boxplot is made from 10 values of the number of disk IOs,

one for each process. We observe from this figure that indeed the bucket epochs contain a variety of ranges of the number of disk IOs per process, with the smallest spans for 1-minute bucket epochs and the largest span for the 10-minute epoch.

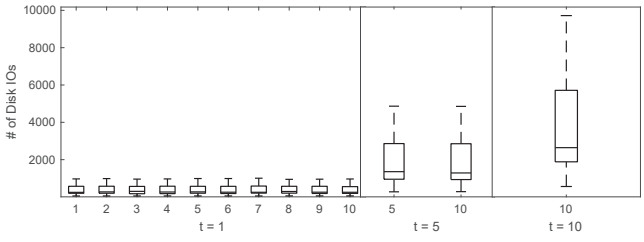


Fig. 2. The number of disk IOs stored in bucket epochs.

Accuracy: To study the overall accuracy, we used $n = 20$ and $m = 5$, and performed three simulations using $t = 1, 5,$ and 10 minutes. From the bucket epochs resulting from each simulation, we queried the total number of disk IOs performed by each of the 10 processes throughout the duration of 10 minutes. Note that for the bucket epochs resulting from simulation using $t = 1$, each query requires the query processing engine to estimate a value from each of the 10 bucket epochs. Similarly, for the bucket epochs resulting from simulation using $t = 5$ and 10, each query requires the query processing engine to estimate a value from two and one bucket epochs, respectively. Figure 3 shows a scatter plot of the actual number of disk IOs vs the estimated number of disk IOs from the bucket epochs resulting from $t = 1$ minute. Due to space limitation, we do not show the scatter plots of the estimates obtained using the bucket epochs resulting from $t = 5$ and 10 minutes, rather only describe the observations.

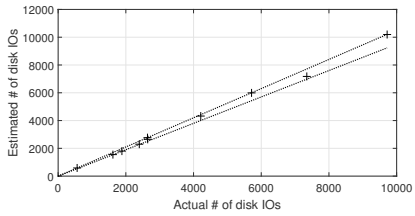


Fig. 3. Actual # of disk IOs vs. estimated # of disk IOs

We observe from this figure that all estimates always lie within the $\pm 5\%$ error lines, *i.e.*, our IoT*m* framework estimated the number of disk IOs for each process with less than 5% error. The estimates from the $t = 5$ and 10 minute wide bucket epochs have smaller error compared to the estimates from $t = 1$ minute wide bucket epochs because for $t = 1$ minute wide bucket epochs, the query processing engine has to obtain 10 estimates from 10 bucket epochs. As each estimate has error, the more the number of estimates involved in answering a query, the higher the error. We make two conclusions from the discussion above. First, IoT*m* accurately estimates the values of the number of disk IOs with less than 5% error. Second, increasing the bucket epoch duration reduces the error. However, note that the increase in bucket epoch duration may not always be feasible because larger duration implies that more information will be inserted into the bucket epochs, and thus, the size b of each bucket will need to be increased, which may not always be feasible for resource constrained IoT nodes.

Effect of the Number of Buckets in Array: To study the effect of n on the accuracy of the IoT*m* framework, we performed 7 sets of the same three simulations described above (*i.e.*, using $t = 1, 5,$ and 10), where each set of simulations used a unique value of n. For all these simulations, we kept m fixed at 5. From the bucket epochs resulting from each simulation, we queried the total number of disk IOs performed by each of the 10 processes throughout the duration of 10 minutes. Figure 4 plots the average relative error in the estimates across the 10 processes for different values of n and different bucket epoch durations. We define *relative error* in an estimate as ($|$`actual value` $-$ `estimated value`$|$/`actual value`). We observe from this figure that as n increases, the relative error decreases. This is intuitive because when the value of n is large, the information of different processes is spread over more diverse sets of buckets, and each bucket is storing information from fewer number of processes. In other words, there is less *noise* in any given bucket when seen from the perspective of any process whose subarray the given bucket is in. We conclude from this discussion that the larger the n, the lower the error. However, very large values of n may not be possible for resource constrained IoT nodes.

Effect of the Number of Buckets in Subarray: To study the effect of m on the accuracy of IoT*m*, we performed 8 sets of the same three simulations (*i.e.*, using $t = 1, 5,$ and 10), where each set of simulation used a unique value of m. For all these simulations, we kept n fixed at 20. Figure 5 plots the average relative error in the estimates across the 10 processes for different values of m and different bucket epoch durations. We observe from this figure that the relative error is a convex function of m. The initial decrease in the relative error with the increase in m is because the increase in the number of buckets in a subarray increases the *observations* from which an estimate is obtained. It is a well-known result in estimation theory that the increase in the number of observations decreases the variance in the error of estimates [14]. The subsequent increase in the relative error is because as m increases, more and more processes start sharing the same bucket, which increases the amount of noise in each bucket and thus increases the error in estimates. We conclude from this discussion that given the values of other parameters, such as n and b, an optimal value of m exists that minimizes the error. In our future work, we plan to theoretically derive the expression to calculate this optimal value of m.

Effect of the Number of Bucket Epochs: To study the effect of the number of bucket epochs across which the query spans on the accuracy of our framework, we performed a simulation using $t = 1$, $n = 20$, and $m = 5$. This simulation resulted in ten 1-minute wide bucket epochs. We executed 10 sets of queries, where each set comprised of 10 queries corresponding to the 10 processes to estimate their number

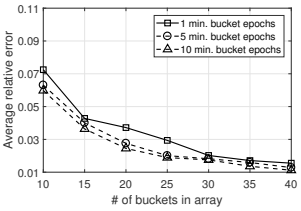


Fig. 4. Effect of n on average relative error ($m = 5$)

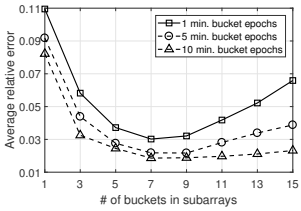


Fig. 5. Effect of m on average relative error ($n = 20$)

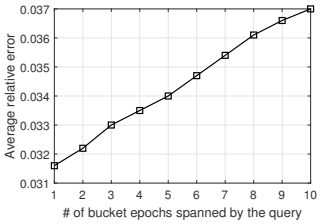


Fig. 6. Effect of bucket epochs spanned by a query on avg. rel. error

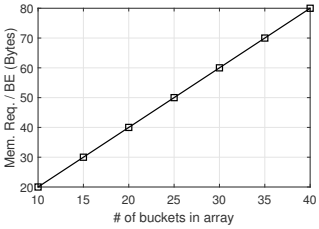


Fig. 7. Memory required as a function of the number of buckets

of disk IOs. Each query for any given process in the i^{th} set of queries ($1 \leq i \leq 10$) was to estimate the number of disk IOs performed by that process in the first i minutes of the 10 minute trace. Thus each query in the i^{th} set spans i bucket epochs. Figure 6 plots the average relative error across all queries in each set of queries. We observe from this figure that as the number of bucket epochs across which the query spans increases, the average relative error increases (although only slightly). This concurs with our earlier observation that using a lager duration for each bucket epoch is advisable as it decreases the number of bucket epochs from which any given estimate is obtained. However, as also discussed earlier, the larger duration may not always be feasible due to the need for increasing the size of each bucket and the limited resources on IoT nodes. The system administrator should first determine how much memory is available for data structure on the IoT nodes and how frequently the measurements will be taken. Based on this, he/she should decide the appropriate size of each bucket and the duration of bucket epochs.

***Memory**:* Figure 7 plots the memory required by our data structure corresponding to the different values of n in Figure 4. Each point in Figure 7 is obtained by multiplying n with $b = 16$ bits. We observe from this figure that even for the highest accuracy configuration (*i.e.*, when $n = 40$), our data structure requires just 80 bytes of memory. Most IoT nodes, such as Raspberry Pi and Photon IO have much higher amounts of RAM available compared to 80 bytes. This shows that our proposed INU can easily be implemented on IoT nodes.

B. Round Trip Latency per Flow

Next, we describe how the IoT*m* framework can estimate the average round trip times (RTT) experienced by the packets in any flow. To measure RTT for any packet, we need two time stamps: one when the packet leaves the IoT node and another when its ACK arrives back. Consider an arbitrary flow with ID f that has l packets. The ID of a flow can be any flow identifier, such as the standard five tuple. Let S_i represent the time stamp when the i^{th} packet of this flow is sent and let R_i represent the time stamp when the ACK of this i^{th} packet arrives back. The average RTT of the packets in a flow f is:

$$\text{RTT}_f = \frac{(R_1 - S_1) + \ldots + (R_l - S_l)}{l} = \frac{1}{l}\left(\sum_{i=1}^{l} R_i - \sum_{i=1}^{l} S_i\right)$$

This equation shows that if we simply add the time stamp of each ACK in our data structure and subtract the time stamp of each sent packet from the data structure, then we will be storing the sum of RTT values of all packets in the data structure. This method, however, will work only if there are no packet losses. To demonstrate the use of our framework for measuring average RTTs per flow, we assume no packet losses. Handling packet losses is straighforward: maintain two bucket arrays for separately storing time stamps of sent and received packet and another two bucket arrays to separately keep a count of the number of sent and received packets.

1) Method: When the control unit instructs INU on any IoT node to start recording RTT values of flows, the INU initializes the bucket arrays **B** and **C**, each comprising n buckets. To make sure that the time stamp of the ACK of a packet is added to the same bucket from which the time stamp of its transmission time was subtracted, every time a packet of a flow ID f is transmitted or an ACK of a packet of that flow ID f arrives, instead of randomly choosing a number j in the range $[1, m]$ and appending it to flow ID before calculating the hash function, INU applies the modulo m operation on the packet sequence number, adds 1 to it, and appends that to the flow ID. It then calculates the hash function $H(f, \text{Seq}\#\%m + 1)$ whose output is uniformly distributed in the range $[1, n]$, and adds the time stamp to the bucket $\mathbf{B}[H(f, \text{Seq}\#\%m + 1)]$ if processing a received ACK or subtracts the time stamp from the bucket $\mathbf{B}[H(f, \text{Seq}\#\%m + 1)]$ if processing a transmitted packet. For each sent packet of any given flow with ID f, it also increments the corresponding bucket $\mathbf{C}[H(f, \text{Seq}\#\%m + 1)]$ by one to keep count of the number of packets of that flow.

To estimate the average RTT experienced by the packets of any given flow with ID f over a desired period of time, the control unit first identifies all bucket epochs whose time frames overlap with that desired period of time. Next, it asks the query processing engine(s) to estimate the sum of RTTs from each identified bucket epoch **B** and the count of the number of transmitted packets from each corresponding bucket epoch **C**. The query processing engine(s) use Eq. (7) to obtain these estimates and send them to the control unit. The control unit then divides the sum of estimates from all **B** bucket epochs with the sum of estimates from all **C** bucket epochs to estimate the average RTT experienced by the packets of the flow f.

2) Traces: To evaluate the accuracy of our framework, we collected traces from a Raspberry Pi executing MQTT protocol and sending/receiving packets from an MQTT broker. The Raspberry Pi ran 50 different processes, each having a unique persistent TCP connection with the MQTT broker. This

resulted in 50 flows with distinct IDs. The flow ID is the standard 5 tuple. Each process sent 100 byte messages to the MQTT server on up to 10 different topics. Furthermore, each process sent messages at different rates: the process with flow ID f_i $(1 \leq i \leq 50)$ carried an average of i messages per second with exponentially distributed inter-arrival times. We used `tcpdump` to log all departing and arriving traffic at the Raspberry Pi for a duration of 10 minutes. The `pcap` files resulting from the `tcpdump` contain the flow ID for each packet as well as the TCP sequence numbers, which the INU uses to decide the bucket in which the time stamp of an ACK should be added and from which the time stamp of a sent packet should be subtracted. Figures 8 and 9 plot the CDFs of the packet counts of the 50 flows and the RTTs experienced by the packets across these 50 flows, respectively. We observe from these figures that we have flows ranging from very few packets to a very large number of packets and latencies ranging from a few milliseconds up to about 50 milliseconds.

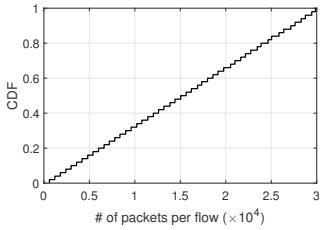


Fig. 8. CDF of packet counts of flows

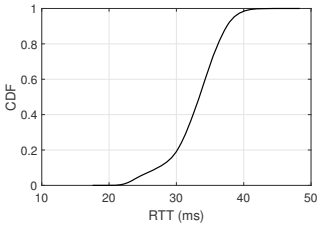


Fig. 9. CDF of RTTs of all packets

3) Evaluation: In all our experiments, we used $b = 32$ bits because time stamp values at the granularity of milliseconds require larger memory compared to the number of disk IO values. Similar to Sections IV-A3, we used our INU emulator, which traverses the `pcap` file. Every time the simulation time matches the time stamp of a packet in the `pcap` file, the INU emulator increments a bucket in the **C** bucket subarray corresponding to the flow ID of that packet, and adds (subtracts) the time stamp of the received ACK (sent packet) from the corresponding bucket in the **B** bucket subarray.

Accuracy: To study the accuracy, we used $n = 20$ and $m = 5$, and performed three simulations using $t = 1, 5,$ and 10 minutes. From the bucket epochs resulting from each simulation, we queried the average RTTs experienced by each of the 50 flows. Figures 10(a), 10(b), and 10(c) show scatter plots of the actual average RTTs of each flow vs. the estimated average RTTs from the bucket epochs resulting from $t = 1, 5,$ and 10 minutes, respectively. We observe from these figures that the estimates always lie within $\pm 7\%$ error error lines when $t = 1$ and within $\pm 5\%$ error lines when $t = 5$ and 10.

Effect of Number of Instances: To study the effect of the number of instances on the accuracy of IoT*m*, we performed 9 more simulations by varying the number of flows from 10 to 50 in steps of 5, and using $n = 20$, $m = 5$, and $t = 10$. In any simulation with i flows (where $i = 5j$ and $j \in [2, 10]$), we randomly selected the i flows out of our 50 flows. Figure 11 plots the average relative error in the estimates of the RTTs of the i flows averaged over 100 runs of simulation with i flows. We observe from this figure that, as the number of flows increase, the error slowly increases. This happens because with increase in the number of flows, more and more processes start sharing the same bucket, which increases the amount of noise in each bucket and thus increases the error in estimates. Nonetheless, the error is still small. This slight loss in accuracy brings significant reduction in the memory requirements compared to the naive approach, as we describe next.

Memory: The memory required by our data structure for measuring average RTT is four times of that required for measuring the number of disk IOs because average RTTs need bucket epoch **C**s in addition to bucket epoch **B**s and the size of each bucket is twice the size of each bucket used for disk IOs. Nonetheless, any low end IoT nodes can still easily implement our INU and the data structure for measuring latencies. Note that if one were to use the naive approach of maintaining a unique counter for each flow, as discussed in Section II-B, one would need 50×2 32-bit counters for the 50 flows compared to just 20×2 32-bit buckets used by our data structure. Thus, among the other advantages mentioned in Section II-B of using our data structure instead of the naive approach, in this particular example, our data structure requires 2.5 times less memory compared to the naive approach. The saving in the memory increases further with the increase in the number of instances, which makes our data structure a lot more scalable compared to the naive approach.

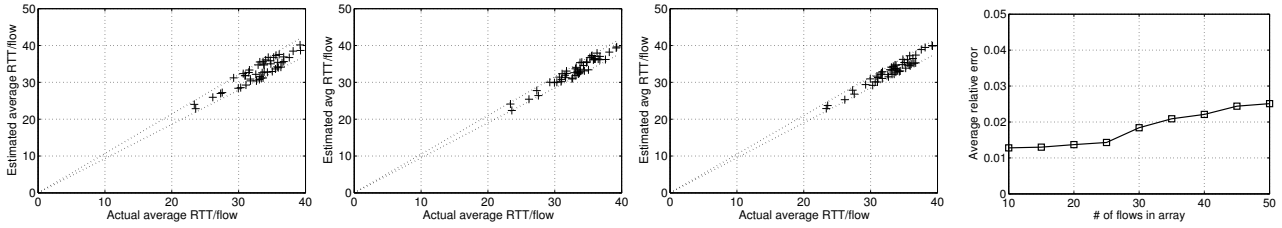
V. Related Work

While the problem of measuring IoT performance metrics has largely been unexplored, work has been done on measuring performance metrics in conventional networks and systems. Next, we first describe a relevant class of data structures, called sketches, which are frequently used to store performance metrics in conventional networks and systems. After that, we describe a recently proposed framework, namely Trumpet [12], designed for fine-grained network monitoring in data centers. Last, we present some representative prior work on measuring performance metrics in conventional networks.

A. Sketches

Count-Min (CM) sketch is the most relevant data structure that can be used to store and estimate the sums and counts of various performance metrics [15]. It has been extensively used in conventional networks and systems [16], [17]. Several other variants of CM-sketch also exist, such as Count sketch [18], conservative update sketch [19], and CM-log-sketch [20].

CM-sketch falls short from two perspectives when measuring IoT performance metrics. First, it requires d hash computations and d memory updates per insertion, which can be too much work for resource constrained IoT nodes. In comparison, our data structure requires just a single hash computation and memory update per insertion. Second, CM-sketch achieves similar error bound as our proposed data structure only if we use a dedicated sketch for each instance. Lets demonstrate this using the number of disk IOs as example.



(a) $t = 1$ minute (b) $t = 5$ minutes (c) $t = 10$ minutes

Fig. 10. Actual average RTT per flow vs. estimated average RTT per flow

Fig. 11. Effect of the number of instances on average relative error

Let s_p represent the number of disk IOs of a process with ID p and let there be N processes in total whose number of disk IOs need to be recorded. We can use a CM-sketch to store the number of disk IOs and obtain the estimate $\tilde{s}_p$ of the number of disk IOs of any process with ID p, as per the method described in [15]. The estimate $\tilde{s}_p$ obtained through CM-sketch satisfies the condition $\tilde{s}_p \leq s_p + \epsilon \times \sum_{j=1}^{N} s_p$ with probability ξ [15]. In comparison, the estimate $\tilde{s}_p$ obtained through our data structure satisfies the condition $\tilde{s}_p \leq s_p + \epsilon \times s_p$ with probability ξ. To achieve similar error bound using CM-sketch, we need to ensure that $\sum_{\forall j} s_p \leq s_p$, which is possible only if we do not add the number of disk IOs of any process other than p to the CM-sketch. Thus, we need a CM-sketch for each process, leading to prohibitively large memory requirements that the resource constrained IoT nodes cannot provision.

B. Data Center Monitoring Framework

Trumpet is a framework similar to IoT*m* in spirit, but designed for measuring performance metrics and detecting events of interest in data centers [12]. It leverages abundance of compute & memory resources and programmability at end hosts to monitor every packet and to report events. Trumpet detects events of interest using triggers at end hosts. It evaluates triggers by inspecting packets at full line rate and reports events to a controller. The fundamental difference between Trumpet and IoT*m* is that Trumpet assumes abundance of compute & memory resources and remote programmability at end host, whereas IoT*m* assumes neither. To avoid requiring abundance of compute & memory resources, IoT*m* delegates decision making about the metrics to monitor and the nodes on which to monitor them to the control unit in CQU. To avoid requiring remote programmability at IoT nodes, IoT*m* employs a generic data structure in INU that can record measurements for a variety of metrics. Thus, IoT*m* is well suited for IoT implementations while Trumpet is well suited for data centers.

C. Other Measurement Schemes

Due to space limitation, we summarize the existing work only on the measurement of one QoS metrics, namely latency, for conventional networks and describe how existing work is infeasible for measuring IoT metrics. Existing schemes for almost all QoS & RU metrics can be broadly divided into two categories: active and passive. Active measurement schemes rely on performing active operations, such as injecting probe packets, to measure the performance metrics. Such schemes are usually easy to implement, but can alter the true value of the metrics due to active operations. Passive schemes do away with active operations but require more computational and memory resources to measure the performance metric.

Active Latency Measurement Schemes: Lee *et al.* proposed MAPLE, a scheme to efficiently store latencies of packets [4]. MAPLE attaches time stamp to each packet at the sender and the receiver calculates the latencies by subtracting the attached time stamp from its current time. Attaching time stamps is the key limitation of MAPLE because it not only requires modifications to standardized packet header formats and data forwarding paths of existing routers and middleboxes but also puts additional work on resource constrained IoT nodes. Furthermore, attaching time stamps can consume up to 10% of the bandwidth [21]. Lee *et al.* proposed RLI, which measures latency of any given flow by inserting time stamped probe packets into the flow [22]. To calculate the latency of the regular packets between two probe packets, RLI applies straight line interpolation. Inserting probe packets is the key limitation of RLI because for fine-grained measurement, the number of probe packets is large and the latency measured with a large number of probe packets significantly deviates from the real latency. The bandwidth that such probe packets waste can actually be utilized for in-band transportation of the measurement data collected by the IoT nodes.

Passive Latency Measurement Schemes: LDA provides passive but aggregate latency measurement between a sender and a receiver [21]. As it does not provide fine-grained latency measurements, it cannot be used to understand the causes of sudden and short-lived deteriorations in the performance of IoT networks. In LDA, both the sender and the receiver maintain a counter vector where each element is a pair of counters: time stamp counter for accumulating packet time stamps and packet counter for counting the number of arriving/departing packets. For each arriving or departing packet, LDA randomly maps the packet to a counter pair in the counter vector and adds the time stamp of the packet to the time stamp counter and increments the packet counter by one. To obtain the aggregate latency estimate, for each counter pair, LDA checks whether they have the same packet counter value and selects all counter pairs that have the same packet counter value for both the sender and receiver. Finally, LDA obtains aggregate average latency by subtracting the sum of time stamps at the sender side from that at the receiver side and divides it with the total number of successfully delivered packets.

VI. Conclusion

The key contribution of this paper is in proposing IoT*m*, a framework for measuring IoT performance metrics, and demonstrating its use and accuracy by applying it to measure various IoT performance metrics. The key technical depth of this paper is in the design and analysis of our generic data structure as well as the estimation theory that enables the query processing engines to accurately estimate the values of the desired metrics. IoT*m* is lightweight in terms of both computational resources and physical memory. It requires just a single hash computation and memory update per measurement and only a few tens of bytes of memory to store several minutes worth of measurements. This makes IoT*m* amenable for implementation on resource constrained IoT nodes. Our experimental results showed that our framework can achieve high accuracy (over 95%) in estimating a variety of IoT performance metrics. In future, we plan to develop theoretical models to calculate the optimal values of the parameters of data structure used in IoT*m*.

Acknowledgment

This work is supported in part by the National Science Foundation under grants # CNS 1616273 and CNS 1616317.

References

[1] "Conserve water with the internet of things," http://www.ibm.com/developerworks/cloud/library/cl-poseidon1-app/.

[2] "Smart buildings: Intel iot platforms open doors to innovation at ibcon," https://blogs.intel.com/iot/2015/06/17/smart-buildings-intel-iot-platforms-open-doors-to-intelligent-buildings-at-ibcon/.

[3] "Smart structures edc - embedded data collector," http://smart-structures.com/technology/EDC-embedded-data-collector/.

[4] M. Lee, N. Duffield, and R. R. Kompella, "A scalable architecture for maintaining packet latency measurements," in *Proc. IMC*, 2012, pp. 101–114.

[5] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Improving accuracy in end-to-end packet loss measurement," vol. 35, no. 4, pp. 157–168, 2005.

[6] M. Hasib, J. Schormans, and T. Timotijevic, "Accuracy of packet loss monitoring over networked cpe," *Communications, IET*, vol. 1, no. 3, pp. 507–513, 2007.

[7] F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot, "On optimal probing for delay and loss measurement," in *Proc. ACM IMC*. ACM, 2007, pp. 291–302.

[8] B. M. Parker, S. G. Gilmour, and J. Schormans, "Measurement of packet loss probability by optimal design of packet probing experiments," *IET communications*, vol. 3, no. 6, pp. 979–991, 2009.

[9] N. G. Duffield, J. Horowitz, F. Lo Presti, and D. Towsley, "Explicit loss inference in multicast tomography," *IEEE Transactions on Information Theory*, vol. 52, no. 8, pp. 3852–3855, 2006.

[10] H. X. Nguyen and P. Thiran, "Network loss inference with second order statistics of end-to-end flows," in *Proc. ACM IMC*. ACM, 2007, pp. 227–240.

[11] Y. Zhao and Y. Chen, "Fad and spa: End-to-end link-level loss rate inference without infrastructure," *Computer Networks*, vol. 53, no. 9, pp. 1303–1318, 2009.

[12] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 129–143.

[13] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-sa publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798.

[14] S. M. Kay, "Fundamentals of statistical signal processing, volume i: estimation theory," 1993.

[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[16] A. Chen, Y. Jin, J. Cao, and L. E. Li, "Tracking long duration flows in network traffic," in *Proc. IEEE INFOCOM*, 2010.

[17] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proc. ACM IMC*, 2004.

[18] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming*. Springer, 2002.

[19] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGMCOMM CCR*, vol. 32, no. 4, 2002.

[20] G. Pitel and G. Fouquier, "Count-min-log sketch: Approximately counting with approximate counters," *arXiv preprint arXiv:1502.04885*, 2015.

[21] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, "Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator," in *Proc. ACM SIGCOMM*, 2009, pp. 255–266.

[22] M. Lee, N. Duffield, and R. R. Kompella, "Not all microseconds are equal: fine-grained per-flow measurements with reference latency interpolation," in *Proc. ACM SIGCOMM*, 2010, pp. 27–38.