INT Based Network-Aware Task Scheduling for Edge Computing

Bibek Shreshta
Computer Science and Engineering
University of Nevada, Reno
Reno, USA
bibek.shrestha@nevada.unr.edu

Richard Cziva
Energy Sciences Network
Lawrence Berkeley National Laboratory
Berkeley, USA
richard@es.net

Engin Arslan
Computer Science and Engineering
University of Nevada, Reno
Reno, USA
earslan@unr.edu

***Abstract*—Edge computing promises low-latency computation for delay sensitive applications by processing data close to its source. Task scheduling in edge computing is however not immune to performance fluctuations as dynamic and unpredictable nature of network traffic can adversely affect the data transfer performance between end devices and edge servers. In this paper, we leverage In-band Network Telemetry (INT) to gather fine-grained, temporal statistics about network conditions and incorporate network-awareness into task scheduling for edge computing. Unlike legacy network monitoring techniques that collect port-level or flow-level statistics at the order of tens of seconds, INT offers highly accurate network visibility by capturing network telemetry at packet-level granularity, thereby presenting a unique opportunity to detect network congestion precisely. Our experimental analysis using various workload types and network congestion scenarios reveal that enhancing task scheduling of edge computing with high-precision network telemetry can lead up to 40% reduction in data transfer times and up to 30% reduction in total task execution times by favoring edge servers in uncongested (or mildly congested) sections of network when scheduling tasks.**

***Index Terms*—Scheduling, In-network Telemetry, P4, Edge-computing**

I. Introduction

Legacy data processing pipelines require edge devices to offload computation tasks to datacenters, however this centralized approach has become the source of performance bottlenecks as increasing numbers of applications started to demand low latency, localized data processing. Edge computing paradigm aims to address this by placing compute resources closer to end devices to reduce communication delay significantly [1], [2]. Edge computing also alleviates network congestion by minimizing the amount of data that needs to be carried out over the network. Due to its benefits, edge computing has been adopted to wide-range of applications including serverless computing [3], distributed computing [4], and big data processing [5].

Despite being located closer to end users and devices, edge computing is not completely immune to network congestion. In particular, when end devices rely on shared networks to submit tasks to edge servers, network congestion will inevitably take place and lead to increased network delay and degraded transmission rates. For example, a recent initiative by FABRIC Testbed [6] and ESnet's High Touch Project [7] aims to deploy compute servers in the network for scientific edge computing, which will require edge computing workloads to share network resources with regular traffic. Therefore, edge server selection decisions cannot only be done on the basis of physical closeness as poor network performance can severely degrade the performance of data transfers between edge servers and end devices. However, dynamic and unpredictable nature of network congestion makes it extremely difficult to accurately measure or predict network conditions at the time of task scheduling. While traditional network monitoring practices (e.g., port-level and flow-level statistics) are sufficient to understand long term behavior of network utilization and flow performances, their reporting frequency in the order of tens of seconds falls short to capture transient congestion events [8], [9]. Simply increasing reporting intervals would also not suffice since neither port-level statistics nor sampled flow-level statistics provide the level of precision needed to understand the severity of network congestion.

In-band Network Telemetry (INT) [10] is a new paradigm to collect fine-grained telemetry information by adding performance metrics to individual packets as they pass through network devices. With INT, one can embed several performance metrics such as device ID, ingress port ID, queue length, queue congestion status, and hop latency to individual packets at switches and routers to gain deep insights into network condition. Therefore, we leverage INT to detect network congestion events timely and precisely such that delay-sensitive workload can be scheduled to edge servers located in uncongested sections of the network. We find that taking network congestion into account when making task scheduling decisions lead up to 30% improvement in task completion times and 40% improvement in transfer times. In summary, this paper makes following contributions:

- We develop a network monitoring framework that combines INT with active probing to achieve high-precision visibility while keeping system overhead at minimum.
- We introduce delay and bandwidth estimation technique based on queue size information.
- We propose a network-aware scheduling algorithm for edge computing workloads to minimize the impact of network congestion in task and transfer completion times.

- We run extensive experiments in an emulated network environment using Mininet to assess the performance of the network-aware scheduling under various workload and congestion scenarios.

II. Background

In this section, we provide brief details for technologies that are leveraged in this work to implement network-aware task scheduler for edge computing.

Programmable data plane: Network devices have two distinct abstractions known as control plane and data plane. While control plane manages packet processing and forwarding decisions, data plane is responsible to execute the rules set by the control plane [11]. Network devices traditionally are equipped with a fixed set of functionalities (e.g., checksum verification and packet forwarding) that can be executed when processing packets, but programmable data plane enables custom actions to be executed such as embedding device status information (e.g., device ID and queue length) to packets for high-precision network monitoring [10].

P4 Programming Language: P4 is a high-level programming language for protocol-independent packet processors [12]. It is used to implement data plane operations with custom match-action processing pipeline, making packet processing generic and extendable with ease. Switches that support data plane programming can be programmed with P4 to define custom packet processing rules. P4 offers four programmable blocks as *Parser*, *Ingress Control Flow*, *Egress Control Flow*, and *Deparser*. In the *Parser* stage, the incoming packets are parsed to extract the protocol information required at stages ahead. For example, the incoming packets are broken down into their protocol specific fields such as TCP and UDP. *Ingress Control Flow* deals with forwarding decisions to be made using forwarding tables. Entries in the table are matched with the packet IP information to assign the correct egress port information. Also custom action execution is possible at this stage. Once packets are ready to be transmitted to the network (i.e., they are at the beginning of the egress queue) *Egress Control Flow* allows another round of executions to be done. In the *Deparser* stage, the processing and manipulation of packets is completed, thus packets are reconstructed by the deparser to be forwarded to the next hop in the network.

In-band Network Telemetry (INT): Port-level and flow-level statistics have been the primary method of monitoring in networks. While these solutions help to uncover major performance problems such as faulty ports or misconfigured routing tables, they fall short to capture transient events [9]. In-band Network Telemetry addresses this limitation by allowing network monitoring to be done at the packet level. It leverages programmable data plane to capture network telemetry data (and potentially add to packets) in network devices at line rate with minimal processing overhead. Fine-granular, real-time network telemetry data obtained with INT has been used to implement many functionalities that were not possible previously including precise performance troubleshooting [13], path tracking [14], and highly-efficient congestion control algorithm design [15], [16].

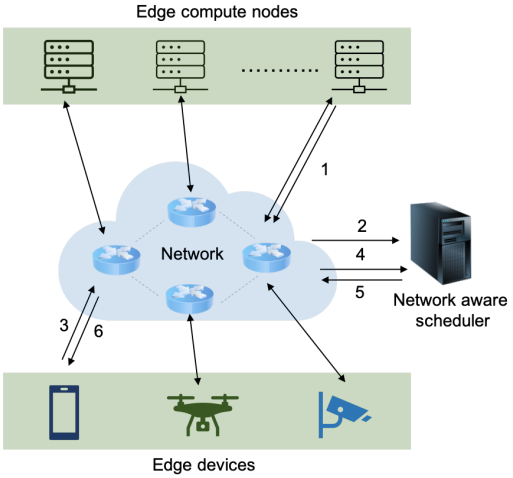

Fig. 1: High-level architecture of proposed network-aware scheduler. It collects INT at packet-level but saves it in switch/router registers. (1) To gather INT stored in network device registers, edge servers periodically send probing packets to the scheduler. (2) The probing packets with INT are received and processed by the scheduler. (3,4) When an edge device wants to schedule a task, (5) it first requests a list of candidate edge servers from the scheduler, then (6) schedules its task(s) to one or more edge servers.

III. Network-Aware Task Scheduler

Figure 1 illustrates the system architecture of the proposed network-aware task scheduler for edge computing. The task scheduler is responsible for directing incoming tasks to distributed edge servers. The task scheduler accepts requests from edge devices and responds back with IP addresses of candidate edge servers. The edge devices then can submit their tasks to edge server(s) directly for processing. As a central command control, the scheduler is responsible for maintaining network status information and selecting edge servers that will satisfy the requirements of tasks while taking network congestion into account.

A. INT Collection

The scheduler relies on data plane processing to gather network telemetry at the packet-level granularity. As P4-enabled devices can execute custom code to process packets when they arrive (i.e., ingress processing) and leave (i.e., egress processing) the device, it allows device statistics to be added to each packet to capture network telemetry at high granularity. However, this comes at the cost of increased overhead since the amount of packet payload reserved for telemetry data will grow quickly as the number of network devices that packets go through increases. As an example, adding only two INT fields to each packet requires 4.2% packet payload to be dedicated for INT data collection when packet traverses five switches [15]. The rate will grow quickly

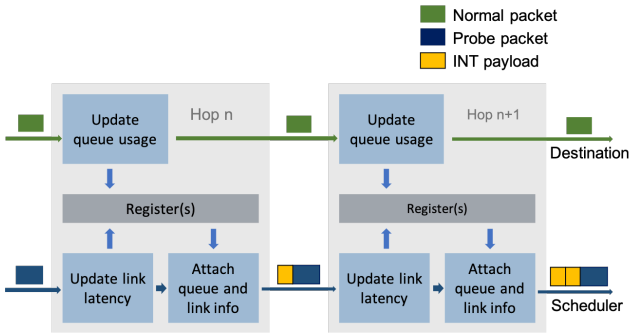

Fig. 2: Data plane processing pipeline.

as the number of INT fields or the number of visited network devices increases. We therefore save telemetry information at device registers and update them as packets are processed. Please note that we do not create a separate register for each packet, rather we create one register for each INT parameter and update its value as new packets are observed. To gather telemetry data stored in network device registers, we schedule probing packets periodically from edge servers to the scheduler which will save telemetry data in its payload as it traverses in the network.

Figure 2 illustrates this process as packets travel through two network devices. When a network device encounters a regular packet, it saves current values of telemetry data (e.g., queue occupancy) to its registers, then forwards the packet to its next hop without any modification. On the other hand, when a network device spots a probing packet, it attaches the telemetry data saved in its registers to the payload of the probe packet before forwarding it to the next hop. The values in device registers are reset to initial value once they are added to the probe packet. In addition to gathering network telemetry data, probe packets are also used to measure link latency. Although latency between network devices is expected to be relatively stable over time, we use probe packets to periodically measure it to capture jitter characteristics. To do so, timestamp information is added to the payload of the probe packet just before it is pushed out of a network device (i.e., at the egress processing stage). When the probe packet arrives at the next hop, the timestamp information is extracted from the packet payload and used to calculate link latency. Note that this extraction is done before the packet is enqueued in the second network device, thus it only measures link latency. The measured link latency information along with other telemetry data is then added to the packet payload as it travels to the scheduler.

As we rely on probe packets to collect telemetry data from network devices, it is important to make sure at least one probe packet travels through each network device at regular intervals. We therefore schedule a probe packet from each edge server to the scheduler at 100ms intervals. While it is possible that probe packets may not travel all devices depending on network topology and edge server distribution in the network, we leave route selection optimization for probe packets as a future work and assume that the probe packets visit each device at least once and gather saved telemetry data. We use UDP with certain IP header fields set (aka Geneve option [17]) to create probe packets such that network devices can distinguish them from regular traffic. Probing packets generate 120Kbps (10 packets/sec $\times$ 1.5KB/packet) network traffic which is a negligible (e.g., only 1.1% when network bandwidth is 10 Mbps) compared to amount of overhead would be introduced by padding INT information to each network packet.

B. Network Mapping

The scheduler serves the queries received from end devices as shown in step 3, 4 in Figure 1 by returning a list of edge servers that can be used to offload tasks. To do so, the scheduler dynamically builds the network topology using telemetry data reported via probe packets. Specifically, it learns which network devices connected to each other by checking the order of INT data in probe packets. For example, if a probe packet contains INT data in S1-S3-S4 order, we can then deduce that S1 and S3 is connected and so does S3 and S4. After parsing INT data, the scheduler also learns about the congestion conditions for each link.

When a query is received from an end device, the scheduler runs graph traversal to extract paths between the end device and candidate edge servers such that it can respond back with a list of edge servers along with expected network performance (i.e., bandwidth and latency). For instance, if there are two edge servers E1 and E2 in the system and an edge device sends a query to the scheduler, the scheduler will calculate the estimated bandwidth and delay metrics for network path from the edge device to E1 and to E2 separately, and send the results back to the edge device to schedule its tasks. The scheduler can use two options when evaluating the eligibility of edge servers for given tasks. In the first one, it sorts the edge servers based on network delay or available bandwidth depending on task type such that the edge device can select the edge server at the top of the list (i.e., the one with the lowest delay or highest available bandwidth) to launch its task(s). In the second one, the scheduler can respond back with (unsorted) list of all edge devices along with their bandwidth and latency information to let edge devices implement a custom selection algorithm. In the following sections, we present the details of the first option (i.e., bandwidth or delay-based ranking) as the second one can easily be extended using the first one.

C. Delay-based Node Ranking

With the help of probe packets, the scheduler keeps track of transmission delay for each link and queue occupancy for each network device interface. This two metric is then used to calculate the delay between any two devices in the network. Let us consider a network consisting of a set of edge nodes E (containing both end devices and edge servers), a set of network devices N, and a set of links L. Assume two edge nodes e_n and e_m where $e_n, e_m \in E$ are communicating with each other. The network packets traverse through arbitrary number of hops (i.e., network devices such as switches) $h_1, h_2, \ldots h_k \in N$ and links $l_1, l_2, \ldots l_k \in L$. Then, one-way delay can be calculated

by $Delay(e_n, e_m) = \sum_{i=1}^{k} delay(l_i) + \sum_{i=1}^{k} delay(h_i)$. Link delay, l_i, (i.e., transmission delay) is relatively straightforward to calculate as we can mark probe packets with egress timestamps in one hop and extract it at next hop to measure the time difference[1]. Note that *Ingress Processing* of P4 allows us to extract the timestamp added by the previous node before the packet is queued in the device for forwarding, thus link latency calculation avoids queuing delay. To calculate hop latency, we rely on queue occupancy as increased queue length typically results in longer queue wait times. We however find that taking average of all queue sizes observed during a probing period leads to inconclusive results. More specifically, even if a network device is running at full capacity, average queue latency returns close to zero as many packets observe empty queue. Therefore, we rely on maximum queue length value to infer hop latency caused by queuing delay.

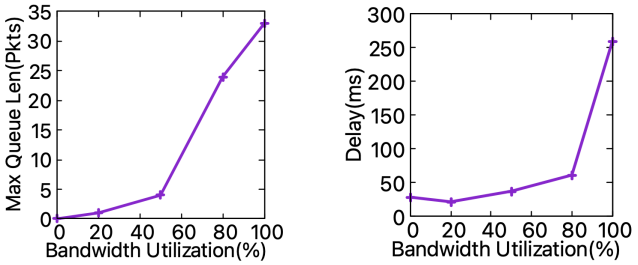

Fig. 3: Max queue length (left) and delay experienced by the packets (right) at different utilization levels.

Figure 3 demonstrates the trend for maximum queue size (in packets) and hop latency for increased egress port utilization. In this experiment, we used *Iperf* to generate fixed-rate traffic volume between two hosts that are connected via P4-enabled switch (i.e., BMv2) in Mininet. For example, 10 Mbps fixed traffic is generated for 50% bandwidth utilization experiment when a network interface has 20 Mbps transmission rate. The link delay is defined as 10ms for each link[2], so round trip time is expected to be 40ms when delay caused by the switch is negligible. Although we did not enforce bandwidth limits between nodes, we observe that maximum transfer speed is limited to 20 Mbps due to data plane programming overhead caused by Mininet[3]. Our data plane processing code measures queue length when an *Iperf* packet is processed by a switch and saves it to the register of the switch if the value is larger than all queue length values observed within a probing interval. We also schedule probe packets in every 100ms to collect maximum queue length information from the switch and reset the register value to 0. In order to calculate end-to-end delay, we run *ping* in the background which measures round trip time between the hosts in one-second intervals. We run each bandwidth utilization value for 300 seconds and report the average values for ping and maximum queue length. We can observe that queue length and end-to-end delay increases significantly as bandwidth utilization is increased. Specifically, while maximum queue length is less than 5 packets until link utilization is below 50%, it increases drastically and becomes more than 30 packets as the link utilization is increased further.

In regard to network delay, ping returns close to 40ms for RTT when the switch is at 0% utilization. As the utilization is gradually increased to 80%, we observe slow but steady increase RTT as it reaches to $50-60$ms. However, when switch port is at full utilization, the delay increases sharply and hits to 250ms, more than $6x$ increase compared to the baseline value. As a result, we can deduce that link utilization has positive correlation between queue size and network delay. Although link utilization is not directly provided by INT, we can exploit the positive relationship between link utilization and maximum queue size to predict the hop latency. Specifically, a conversion factor k is introduced to translate observed queue sizes to hop latency. We observe that the conversion factor of $k = 20$ms is sufficient to identify major congestion events in our experiments, we leave its automation and fine-tuning as a future work.

Algorithm 1: Ranking algorithm to sort edge servers based on network delay with respect to edge node.

Result: Set of edge nodes ranked by delay (N)
e_n = Edge node initiating the query ;
G = Graph representation of the network ;
$E(G, e_n)$ = Edge nodes reachable from e_n in G ;
$L(e_n, e_i)$ = Get links between given edge nodes ;
$H(e_n, e_i)$ = Get hops between given edge nodes ;
$D(l_i)$ = Delay in the link l_i ;
$Q(h_i)$ = Max queue occupancy of hop h_i ;
$S(A)$ = Sort the given array A by delay ;
N = [ ] (Array of result nodes initially empty) ;
k = Queue occupancy to latency conversion factor ;
foreach $e_i \in E(G, e_n)$ **do**
  $totalLinkDelay = 0$;
  foreach l_i *in* $L(e_n, e_i)$ **do**
    $totalLinkDelay \mathrel{+}= D(l_i)$;
  end
  $totalHopDelay = 0$;
  foreach h_i *in* $H(e_n, e_i)$ **do**
    $totalHopDelay \mathrel{+}= k * Q(h_i)$;
  end
  $\Delta = totalLinkDelay + totalHopDelay$;
  $N \Leftarrow N + [(e_i, \Delta)]$
end
$N \Leftarrow S(N)$;
return N;

Now that we can infer delays between the edge nodes, we use Algorithm 1 to sort available edge nodes based on latency from the end device stand point. The algorithm first calculates

[1]BMv2 switches used in the experiments are synced using Network Time Protocol (NTP)

[2]Although we set link delays to 10ms in the experiments, our analysis with smaller link delays also return similar results.

[3]Please note that production switches that support data plane programming do not experience a similar throughput slowdown when P4 programs execute, so observed limitations in Mininet experiments are solely because of BMv2 switch implementation.

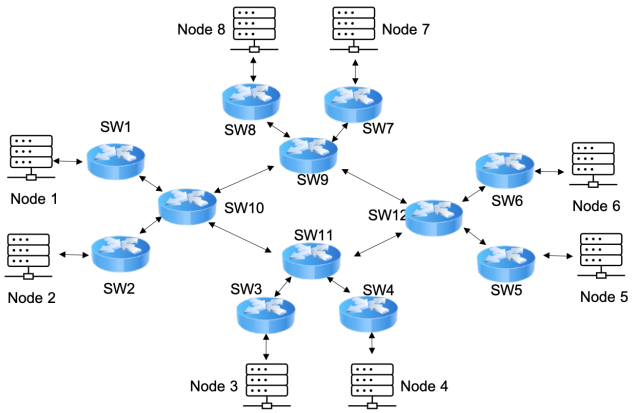

Fig. 4: Experimental topology. Node 6 is dedicated as the scheduler whereas all other nodes are used to schedule tasks as edge device or execute tasks as edge server.

routes between the end device, e_n and candidate edge servers, E. Then, it calculates transmission delay for end-to-end path by summing up link delay for each individual link in the route. Finally, it predicts hop latency introduced by each network device based on maximum observed queue size in the last probing interval and adds those values to transmission delay to calculate end-to-end delay.

D. Bandwidth-based Node Ranking

In this ranking method, we leverage the relationship between the maximum queue utilization and corresponding bandwidth utilization as presented in Figure 3 to infer the level of network utilization. We then estimate available bandwidth for each link between end device and edge servers. Let us consider two nodes where $e_n, e_m \in E$ are communicating with each other. The packet traverses through links $l_1, l_2 \dots l_k \in L$ each with available bandwidth $b_1, b_2 \dots b_k$. Then, throughput of a transfer between these nodes can be estimated to be the minimum available bandwidth among all the links represented as $throughput(e_n, e_m) = min(\{b_1, b_2 \dots b_k\})$. The scheduler estimates transfer throughput for all candidate edge servers with respect to the end device, then sorts them in descending order before responding back to end devices with a list.

IV. Experiment Analysis

The experiments are carried out in Mininet [18], a network emulator that is capable of emulating large networks. We used Mininet Cluster to be able to increase the scale of the experiment by utilizing multiple physical machines. Behavioral Model (BMv2) switch [19] is used to connect nodes, which is a standard reference P4 software switch capable of running P4 programs required in our experiments. Four servers with 32 GB RAM and 4 Core CPU running Ubuntu Server 18.04 were used along with a HP Procurve switch to provide physical connectivity. Figure 4 illustrates the topology used in the experiments, where a total of 8 nodes are connected via 12 switches.

In the experiments, *Node 6* is designated as the scheduler, which is, in addition to responding to the scheduling request, responsible for handling probing packets to learn about network congestion. All the other nodes periodically sends probe packets to gather INT data that is saved in switch registers. All nodes including the scheduler can submit tasks as edge device and execute tasks (unless they are the submitter) as edge server. We compared the INT-based network-aware scheduler with two alternative scheduling solutions as *Nearest* and *Random*. In the nearest node scheduling, a node always submits tasks to the closest node to achieve low latency communication. Since all links have the same 10ms delay, nodes that are located three hop away are the nearest node for each other. For example, Node 7 and Node 8 are the nearest nodes for each other. For the sake of simplicity, we assume that nearest nodes are calculated ahead of time, so no runtime network topology mapping is required for this method. In the random scheduling approach, tasks are scheduled to randomly selected nodes in the network to achieve load balancing.

Type	Data Size (KB)	Execution Time (ms)
Very small (VS)	0 - 1000	0 - 2000
Small (S)	1500 - 2500	2500 - 4500
Medium (M)	3000 - 4000	5000 - 7000
Large (L)	4500 - 5500	7500 - 9500

TABLE I. Data size and execution times defined for different workload sizes used in the experiments.

We simulated two types of workloads as serverless computing and distributed computing to represent scenarios where the use of edge computing is beneficial to optimize task processing. For example, keeping computation close to the data source has potential to greatly reduce task completion time for Function-as-a-Service type workloads in which a significant portion of turnaround time is spent during network communication. Distributed computing workload represents application scenarios where multiple tasks are created to complete a job such as distributed/federated machine learning training. We configured serverless computing jobs to submit one task and distributed computing workload jobs to submit three tasks. These two workloads were experimented on varying size of data transfers and execution time as shown in Table I. We injected background traffic to the network using *Iperf* to simulate network congestion. At any given time, one or two Iperf transfers run between randomly selected nodes for 30s or 60s duration. Thus, different regions of the network becomes congested during the experiments. Although nodes to run background traffic and submit tasks are selected randomly, we used the same order when comparing different scheduling algorithms to ensure fairness. Each experiment consists of 200 tasks and we present average results for each task type by taking the average of all tasks in same category as listed in Table I.

A. Delay-based Ranking

Figure 5 shows the comparison of scheduling algorithms for serverless computing workload when network-aware scheduling uses delay as a metric when ranking edge servers. We

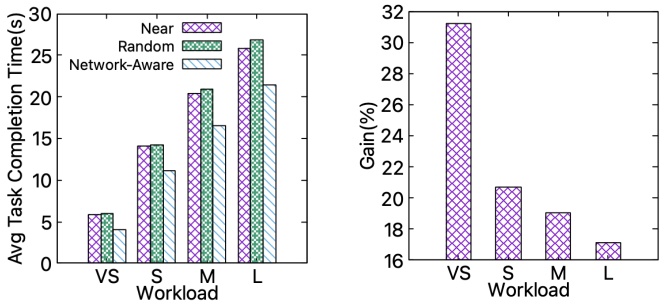

Fig. 5: Serverless computing workload experiment using delay-based node ranking for network-aware scheduling: Average task completion time (left), performance gain (right) for very small (VS), small (S), medium (M) and large (L) tasks.

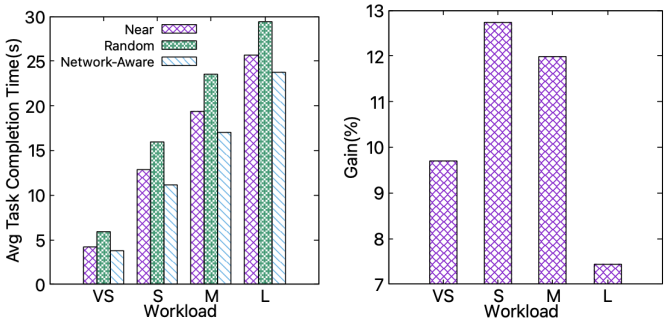

Fig. 6: Distributed computing workload experiment using delay-based ranking for network-aware scheduling: Average task completion time (left), performance gain (right) for very small (VS), small (S), medium (M) and large (L) task types.

observe that the performance gain of network-aware scheduler over the nearest-node selection strategy ranges between $17-31\%$ depending on the task size. The maximum gain is achieved for very small workload which indicates that latency-based task scheduling is most helpful when execution time and data size is small. For distributed computing workload, three nodes are selected to offload tasks. As can be seen in Figure 6, network-aware scheduling again outperforms nearest-node and random selection schemes. The performance gain over the nearest-node scheduling ranges between $7-13\%$. It is important to note that large tasks yields the lowest performance gain for both types of workloads, which can be attributed to the fact that network congestion affects smaller workloads more than it affects larger ones.

B. Bandwidth-based Ranking

We also assessed the performance of gain of network-aware scheduling for distributed computing workload when available network bandwidth is used to sort candidate edge servers. Since distributed computing jobs consist of three tasks, the scheduler will sort the candidate edge servers based on available bandwidth between edge devices and edge servers. Note that even though both latency-based and bandwidth-based node selection is positively correlated with amount of traffic on network devices, latency based selection is likely to select lightly congested nearby nodes over uncongested distant

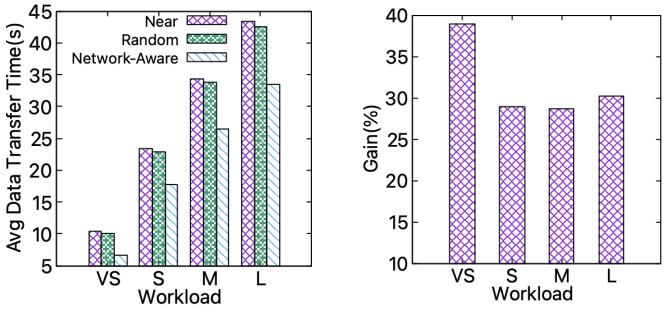

Fig. 7: Average execution times (left) and performance gain (right) for data transfers in distributed computing workload experiment using bandwidth-based ranking for network-aware scheduling.

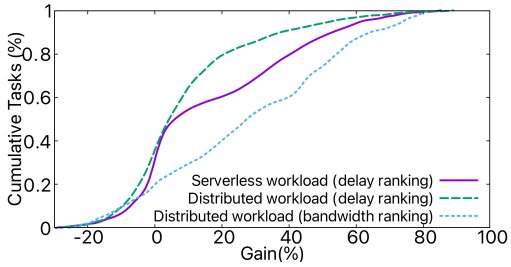

Fig. 8: Empirical cumulative distribution function of the number of tasks executed according to the performance gain observed in task completion time.

nodes to keep transmission delay as small as possible. On the other hand, bandwidth-based selection can prefer remote nodes as long as the available network bandwidth from end device to remote edge servers is higher compared to nearby edge servers. Figure 7 presents the performance of scheduling algorithms in terms of time taken for each workload to transfer data from end device to edge server. We see that transfer time can be reduced by $28-40\%$ with the help of bandwidth-based tasks scheduling. In terms of task completion time, $22-35\%$ reduction is observed with the highest reduction observed for very small workloads[4]. Although the highest performance gain is still observed for small tasks, transfer times decreased considerably (nearly 30%) for large workloads as well.

Figure 8 demonstrates the distribution of performance gain for task completion compared to nearest-node selection strategy. We observe that 19% of tasks for distributed computing workload when using bandwidth-based ranking and 38% of tasks when using delay-based ranking experience zero or negative gain when network-aware scheduling is employed. This is mainly because of measurement jitter that causes the network-aware scheduler to make suboptimal decisions especially when network is lightly congested. Specifically, probing packets can detect small queue build up in network devices even when network congestion is negligible, causing de-prioritization of nearest nodes. On the other hand, more than 60% of distributed computing tasks experience 20% or higher reduction in execution time when using network-aware scheduling with bandwidth-based ranking. Similar 20% or

[4]Figures for task completion times are not presented due to space limitations.

higher decrease in task execution time is observed for 40% and 20% of tasks in serverless and distributed workloads when using delay-based ranking is used. Even further, $10 - 20\%$ of all tasks achieves more than 60% smaller execution times using network-aware scheduling.

C. Impact of Probing Frequency

In this section, we assess the impact of probing frequency on the performance of network-aware scheduling. Specifically, we evaluate the interval between probing packets that are used to gather INT data from switches. Note that this does not affect how INT is collected at the switches but rather changes how often the scheduler receives updates from network devices. Our hypothesis is that higher probing frequency has a better chance to detect dynamic changes in the network, thereby paving the way for better performance. We choose five probing intervals to evaluate as 0.1s (the default value), 5s, 10s, 20s, and 30s (typical SNMP monitoring interval).

We use distributed workload and run experiments under two different scenarios. In the first scenario, we use a medium workload size and injected background traffic, *Traffic 1*, that changes less frequently. For the second scenario, we use a small workload size and inject background traffic, *Traffic 2*, that changes more frequently. In *Traffic 1* experiments, we run three *Iperf* transfers between randomly selected nodes for 30 seconds followed by 30 second sleep. Transfers started with 10 seconds gap in between to ensure that the degree of background changes over time. In *Traffic 2* case, we again run three transfers between randomly selected nodes, but this time transfer duration is 5 seconds followed by 5 seconds of sleep to emulate more dynamic background traffic conditions.

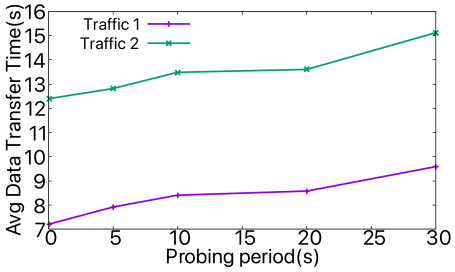

Fig. 9: Impact of probing period on average data transfer time under infrequent (Traffic 1) and frequent (Traffic 2) background traffic fluctuation scenarios.

The results as given in Figure 9 show that low probing intervals lead to lower overall transfer times for both background traffic scenarios, indicating that higher probing frequency increases the likelihood of capturing the subtle changes in the network. As an example, while it takes 12.5s to finish transfers when probing interval is 0.1s, it increases to over $15s$ when 30s probing interval is used, over 20% difference. This outcome further bolsters our motivation to use In-band Network Telemetry over less frequent and coarse-grained network measurement approaches such as SNMP and NetFlow to gather telemetry data.

V. Related Work

Task Scheduling in Edge Computing: Task scheduling plays a critical role in the performance of edge computing as suboptimal scheduling decision could cause significant performance degradation due to increased communication and computation times. Rashidi et al. [20] proposed an adaptive neuro-fuzzy inference system to predict the availability of network and compute resources for task scheduling in edge computing. Li et al. [21] observed that caching on edge servers can significantly improve the performance of latency-sensitive tasks, thus they proposed a cache-aware task scheduling method that increases performance of scheduled tasks in terms of cache-hit ratio, data locality, task response time, etc. Jalaparti et al. [22] has proposed Corral that makes use of characteristics of future workloads to implement scheduling in clustered environment. Cziva et al. [23] adopted optimal stopping theory and integer linear programming to perform latency-optimal allocation of edge resources. Chen et al. [24] adopted a game theoretic approach for achieving efficient computation offloading in a distributed manner in mobile-edge cloud computing environment. Zhao et al. [25] have introduced a cooperative scheduling scheme between the edge and internet cloud to improve the quality of service of mobile cloud computing. Alfakih et al. [26] applied reinforcement learning to optimize task scheduling in mobile edge computing for reduced power consumption.

In-band Network Telemetry: In-band Network Telemetry (INT) allows packet-level network monitoring at line rate. Typical implementation of INT involves capturing and saving status of network devices (e.g., ingress/egress queue size) in packet headers which is then extracted the at the end hosts (or last P4-capable network device) for analysis. Yuliang et al. [15] used INT to build High Performance Congestion Control (HPCC) that improves flow completion times by 95%. Lim et al. [27] showed that INT can be used to improve network load balancing by detecting congestion events quickly and precisely. Pelle et al. [28] proposed telemetry-driven serverless architecture for latency-sensitive edge computing. Despite its benefits, collecting INT data comes at the cost of increased storage cost for network packets since the amount of space needed to store INT data grows quickly as most packets travel through many network devices before reaching to destination. To address this problem, Basat et al. [16] proposed Probabilistic In-band Network Telemetry (PINT) to reduce the number of INT fields to store in each packet while achieving high-accuracy measurements . Vestin et al. [29] designed a programmable event detector for INT which reports only selected event information to the monitoring system to reduce the overhead of INT data collection. Similarly Kim et al. [30] showed that selectively monitoring a certain ratio of packets based on the frequency of changes in the network information can significantly reduce the network overhead and monitoring engine load.

VI. Conclusion and Future Work

In-band Network Telemetry (INT) provides a mechanism to extract high precision network telemetry directly from the data plane at the line rate. This information provides crucial insight into network conditions with high accuracy. In this paper, we propose a network-aware scheduler for edge computing scheduling that leverages INT to gather high-precision network telemetry and incorporates it into edge server selection algorithm. We introduced a novel INT data collection scheme that requires no INT data to be embedded to production traffic packets. We rather save and update INT data in network device registers and periodically collect them using custom probing packets. The experimental results using various workload scenarios show that network-aware task scheduling leads up to 40% reduction in average transfer times and up to 30% reduction in average task completion times.

As a future work, we will extend the network-aware scheduler with compute-aware scheduler to take the availability of compute nodes into account when choosing edge servers. We will also consider heterogeneous edge server scenario in which tasks may have certain hardware (e.g., GPU) or software (e.g., Keras) requirements that needs to be considered when scheduling tasks to edge servers. Moreover, we will investigate the possibility of using network devices to store link utilization and transmission delay information such that end devices do not have to communicate to a central controller receive a response to their scheduling requests.

Acknowledgement

This work is supported in part by the NSF grant 2019164.

References

[1] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020.

[2] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *IEEE Global Communications Conference (Globecom)*, 2020.

[3] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2019, pp. 1–10.

[4] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, and C.-T. Lin, "Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment," *IEEE Access*, vol. 6, pp. 1706–1717, 2017.

[5] Q. Wang, B. Lee, N. Murray, and Y. Qiao, "Mr-edge: a mapreduce-based protocol for iot edge computing with resource constraints," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2019, pp. 1–6.

[6] "FABRIC," https://fabric-testbed.net/, 2021.

[7] R. Cziva, B. Mah, Y. Kumar, and C. Guok, "ESnet high touch services," Supercomputing 2020 - SCinet, 2020.

[8] L. Tan, W. Su, W. Zhang, J. Lv, Z. Zhang, J. Miao, X. Liu, and N. Li, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[9] N. Van Tu, J. Hyun, and J. W.-K. Hong, "Towards onos-based sdn monitoring using in-band network telemetry," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 76–81.

[10] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, vol. 15, 2015.

[11] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–7.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] J. A. Marques, M. C. Luizelli, R. I. T. da Costa Filho, and L. P. Gaspary, "An optimization-based approach for efficient network monitoring using in-band network telemetry," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–20, 2019.

[14] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "Int-path: Towards optimal path planning for in-band network-wide telemetry," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 487–495.

[15] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 44–58.

[16] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "Pint: probabilistic in-band network telemetry," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 662–680.

[17] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwan, D. Daly, M. Hira, and B. Davie, "In-band network telemetry (int)," https://p4.org/assets/INT-current-spec.pdf, 2016, accessed on 2021-02-14.

[18] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[19] "Behavioral-model (bmv2)," https://github.com/p4lang/behavioral-model, 2021.

[20] S. Rashidi and S. Sharifian, "Cloudlet dynamic server selection policy for mobile task off-loading in mobile cloud computing using soft computing techniques," *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3796–3820, 2017.

[21] C. Li, J. Tang, H. Tang, and Y. Luo, "Collaborative cache allocation and task scheduling for data-intensive applications in edge computing environment," *Future Generation Computer Systems*, vol. 95, pp. 249–264, 2019.

[22] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 407–420, 2015.

[23] R. Cziva, C. Anagnostopoulos, and D. P. Pezaros, "Dynamic, latency-optimal vnf placement at the network edge," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 693–701.

[24] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[25] T. Zhao, S. Zhou, X. Guo, Y. Zhao, and Z. Niu, "A cooperative scheduling scheme of local cloud and internet cloud for delay-aware mobile cloud computing," in *2015 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2015, pp. 1–6.

[26] T. Alfakih, M. M. Hassan, A. Gumaei, C. Savaglio, and G. Fortino, "Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on sarsa," *IEEE Access*, vol. 8, pp. 54 074–54 084, 2020.

[27] J. Lim, S. Nam, J.-H. Yoo, and J. W.-K. Hong, "Best nexthop load balancing algorithm with inband network telemetry," in *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE, 2020, pp. 1–7.

[28] I. Pelle, F. Paolucci, B. Sonkoly, and F. Cugini, "Telemetry-driven optical 5g serverless architecture for latency-sensitive edge computing," in *2020 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE, 2020, pp. 1–3.

[29] J. Vestin, A. Kassler, D. Bhamare, K.-J. Grinnemo, J.-O. Andersson, and G. Pongracz, "Programmable event detection for in-band network telemetry," in *2019 IEEE 8th international conference on cloud networking (CloudNet)*. IEEE, 2019, pp. 1–6.

[30] Y. Kim, D. Suh, and S. Pack, "Selective in-band network telemetry for overhead reduction," in *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. IEEE, 2018, pp. 1–3.