University of Nevada, Reno

Network-Aware Task Scheduling for Edge Computing

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Computer Science and Engineering

by

Bibek Shrestha

Engin Arslan, Ph.D. - Thesis Advisor
December, 2021

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

BIBEK SHRESTHA

entitled

Network-Aware Task Scheduling for Edge Computing

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Engin Arslan, Ph.D.
Advisor

Shamik Sengupta, Ph.D.
Committee Member

Scotty Strachan, Ph.D.
Graduate School Representative

David W. Zeh, Ph.D., Dean
Graduate School

December, 2021

Abstract

Edge computing promises low-latency computation by moving data processing closer to the source. Tasks executed at the edge of the network have seen a significant increase in their complexity. The demand for low-latency computation for delay-sensitive applications at the edge is also increasing. To meet the computational demand, task offloading has become a go-to solution where the edge devices offload tasks in part or whole to the edge servers via the network. But the performance fluctuations of the network largely influence the data transfer performance between edge devices and the edge servers, which negatively impacts the overall task execution performance. Hence, monitoring the state of the network is desirable to improve the performance of task offloading at the edge. However, networks are usually dynamic and unpredictable in nature, particularly when the network is being used by multiple other devices and applications simultaneously, resulting in data flows competing with each other for the resources.

In this study, we are leveraging In-band Network Telemetry (INT) to collect fine-grained network information to introduce network awareness in task scheduling for edge computing. Legacy methods of network monitoring that rely on flow-level and port-level statistics are often limited by their collection frequency which is typically in the order of tens of seconds. In contrast, INT can improve the collection frequency by working at the line rate and granularity of information by capturing network telemetry at packet-level directly from the data plane. Such capabilities enable the detection of subtle changes and congestion events in the network, thereby increasing the network visibility while making it more accurate. We implemented a network-aware task scheduler for edge computing that uses high-precision network telemetry for task scheduling. We experimented with different workloads under various congestion scenarios to assess the impact of our network-aware scheduler on the task offloading performance. We observed up to 40% reduction in data transfer time and up to 30% reduction in the overall task execution time by favoring edge servers in uncongested or relatively less congested areas of the network when scheduling the tasks. Our study shows that network visibility is an important factor that can improve task offloading performance. The results so obtained supports our motivation to use INT for obtaining fine-grained high-precision network telemetry to create a network-aware task scheduler for edge computing.

Dedication

This thesis is dedicated to my precious family, who have been a constant source of my inspiration and support. I want to thank all my family members for supporting my decisions and motivating me to help me be where I am today. During this time, my brother took all the burdens onto himself, helping me spread my winds. You are a real source of my inspiration.

I also would like to dedicate this thesis to my friends who have supported me throughout this journey, motivating me at every step.

Acknowledgments

I want to express my sincere gratitude to my advisor, Dr. Engin Arslan, who has provided valuable support, opportunities, guidance, feedback to help me get better both in academics and research.

Also, I would like to extend my thanks to my thesis committee members, Dr. Shamik Sengupta and Dr. Scotty Strachan, for taking the time to review this thesis despite the busy schedules and providing valuable suggestions and guidance to improve my research.

Finally, I would like to thank my family, including my significant other who has always been there for me. It would not have been possible without you all by my side.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application-Specific Integrated Circuit. 7
Bmv2	Behavioral Model. 10, 25, 29
GPU	Graphics Processing Unit. 41
INT	In-band Network Telemetry. i, ii, viii, 3, 4, 8, 10–19, 21, 22, 26, 31, 38–41
IoT	Internet of Things. 1, 5, 6
IP	Internet Protocol. 18
JSON	JavaScript Object Notation. 10
MEC	Mobile-Edge Computing. 11, 12
NTP	Network Time Protocol. 19
P4	Protocol-Independent Packet Processors. 9
PINT	Probabilistic In-band Network Telemetry. 13
PSA	Portable Switch Architecture. viii, 9, 10
RTT	Round-Trip Time. 26

SDN	Software-Defined Networking. 7, 29, 36
SNMP	Simple Network Management Protocol. 2, 3, 10, 21, 38–40
UDP	User Datagram Protocol. 16, 18
VSS	Very Simple Switch. viii, 8

Chapter 1

Introduction

Smart devices and IoT have taken over our society by storm. These devices are ubiquitous and have found their use cases in every household, smart city, etc. Advancements in technology such as 5G has made it possible to have high-capacity, low-latency connectivity among such devices. Such advancements have given rise to numerous other applications of smart devices such as fleet management, industrial automation, etc. At the same time, software applications running on these devices have become more and more complex, demanding more compute resources and requiring latency guarantees [2]. Offloading tasks is one of the solutions used to meet the computational demands of such applications.

In the legacy data offloading paradigm, the edge devices offloaded their tasks to the datacenters to meet the computational demands of the applications. This paradigm comes with a drawback as the data required for computation has to be transferred to the datacenters located far away from the edge devices, which will increase the overall latency due to an increase in communication delay, significantly reducing performance. This is a dealbreaker, especially as the increasing number of applications are demanding low-latency computation. Edge computing solves this problem by bringing the computation closer to the source. Communication delay is vastly reduced by being closer to the source, allowing faster data transfer time between the device and the compute nodes [3]. Reduced congestion due to this proximity also reduces the amount of data that needs to be transferred overall. This results in reduced overall latency of computation as demanded by the applications at the edge [4].

However, just having closer proximity does not mean an end to all of the problems. Though the proximity improves the network conditions in comparison to communicating and offloading tasks to a remote datacenter, congestion still can significantly affect the performance of edge computing workloads. This is because the network is usually shared among many other devices and services that are actively utilizing resources simultaneously. Such a situation can create contention of network resources between edge computing traffic and regular traffic leading to increased congestion. Such congestion can increase the network latency, thus degrading the data transfer performance during task offloading significantly. Such issues can potentially create problems for initiatives taken by FABRIC Testbed [5] and ESnet's High Touch Project [6] in the field of scientific edge computing for research. Scientific edge computing workloads will share the same underlying resources of the research network with existing regular traffic, potentially causing performance issues in such high-profile initiatives.

Edge devices offloading tasks to the edge servers or edge compute nodes would require selecting one or more servers to offload the tasks. Due to the reasons mentioned above, it is not always optimal to select the edge nodes based on the physical proximity. Also, always selecting nearest nodes for task offloading would result in high network resource usage at the edge of the network resulting in poor data transfer performance between edge devices. In order to overcome this problem, it is desirable to monitor subtle changes in the network so that an optimal decision about edge server selection can be made during task scheduling. But monitoring the network at such a fine-granular level and accuracy is easier said than done due to its unpredictable and dynamic nature. Technologies such as Simple Network

Management Protocol (SNMP) [7], NetFlow [8], etc can be used for the purpose of network monitoring at port or flow-level. However, these solutions have lower reporting frequency which is usually in the order of tens of seconds. Hence, these solutions cannot capture subtle and transient network changes and congestion events required to make precise inferences about the network status.

In-band Network Telemetry (INT) is a novel method of collecting network metrics at the packet level. Network devices with programmable data plane can attach fine-grained network performance metrics to the packets as it traverses through those devices. Information such as device ID, ingress/egress port, queue usage, hop latency, etc., can be directly embedded to any target packet using INT. Since INT works at the line rate, the metrics can be obtained at really high frequency. This increases our ability to detect subtle changes in the network and make precise inferences about the network conditions that the scheduler can use to make optimal task scheduling decisions. In this study, we use INT as a network monitoring paradigm to build a network-aware scheduler to schedule tasks at the edge optimally [9]. We compare different types of edge computing workloads under different background traffic conditions to see how our method of edge node selection performs against other baseline selection strategies.

Following are the key contributions of this study.

- We devise a network monitoring paradigm combining INT along with an active probing mechanism to achieve high-precision network visibility while keeping the system overhead at a minimum.

- We introduce techniques to estimate network path delay and bandwidth usage based on the queue usage information of each device.

- We propose a network-aware scheduling algorithm based on INT for edge computing to optimally schedule tasks by avoiding potentially congested paths to minimize data transfer completion times.

- We evaluate the performance of our network-aware scheduling method through extensive experiments using various workload types and congestion scenarios.

This thesis is organized into various chapters. Chapter 2 provides the background on edge computing, programmable data planes, P4 programming language, and In-band Network Telemetry (INT). It also explores related literatures in the field of task scheduling for edge computing and INT. Chapter 3 goes into the details of the proposed network-aware task scheduler exploring the inner working of the scheduler in different phases, and also provide details on node ranking methods used. Details regarding the experiments and outcome analysis are provided in chapter 4. Finally, the conclusion of the study and possible future works are discussed in chapter 5.

Chapter 2

Background and Literature Review

2.1 Edge Computing

The amount of data generated at the edge of the network has been growing exponentially, fueled by the explosive growth and adoption of intelligent IoT devices. According to the study done by IoT Analytics, the number of IoT devices connected to the network is expected to reach 12 billion by 2021 and 27 billion by 2025 [10]. To keep things in perspective, the current population of the earth is about 7.7 billion. That is equivalent to having nearly two connected IoT devices per person in 2021. These devices have various use cases in smart homes, automation, logistics, health industry, smart city, gaming, etc. Newer connectivity technology such as 5G drastically reduces the communication latency providing a suitable environment for IoT [11]. This is expected to push the adoption of IoT devices to greater heights. The amount of data generated by such a massive number of connected devices is astronomical.

Cloud computing has been the go-to solution when additional compute resources are required for performing certain tasks which the local machines are not capable of providing.

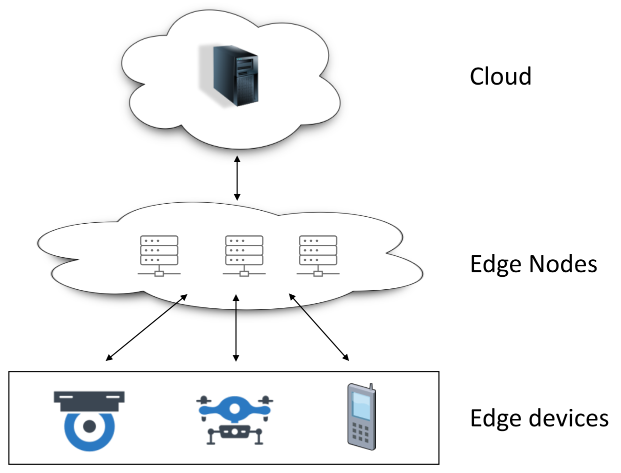

Figure 2.1: Overview of edge computing.

Cloud resources usually reside on the data centers that are multiple hops away from the edge of the network and possibly in different physical regions. Hence, the data needs to travel a much longer distance, which adds to the overall computation latency. A study of network round-trip latency in Azure between different data center regions shows that the latency increases as the distance between the data center increase [12]. This shows that the application requiring real-time or near real-time data will suffer from higher latency to reach the cloud. Considering the enormous amount of data generated by the connected IoT devices at the edge, it would be efficient and desirable to have the processing capability at the edge of the network. This boosts the performance of the edge applications and saves resources wasted due to transferring the data to the remote data centers. Edge computing is a paradigm where the compute resources are closer to where the data is generated at the edge of the network. Figure 2.1 shows the overview of edge computing where the edge devices are closer to the edge nodes than the cloud.

Although all IoT applications benefit from edge computing, the applications that require latency guarantees are the ones that benefit the most. For example, a security camera device with real-time object detection might not have enough computational resources on board and needs to offload the task to the compute nodes. However, task offloading comes with an added cost of communication, adding uncertainty to the overall processing latency. This means that the device might no longer guarantee real-time processing if it uses cloud offloading. On the other hand, edge compute nodes at the edge of the network have lower network latency than the cloud. Offloading to the edge device would maximize the chance of meeting the desired latency guarantee and hence real-time object detection.

Edge computing has seen its use in wide-range applications in areas such as distributed computing [13], big data processing [14], serverless computing [15], etc. that benefits from the proximity of computational resources. Although edge computing can bring computation resources closer to the data source, these resources are usually limited. In comparison, cloud computing can virtually provide unlimited resources on demand which is impossible in edge computing. Hence, not all applications and workloads can make use of edge computing.

2.2 Programmable Data Plane

Networking devices can be divided into two distinct planes: the control plane and the data plane. The Control plane decides the path that the incoming packet will take. On the other hand, the packets are actually forwarded through the data plane. Traditional networking devices are fixed functionality devices, making them extremely inflexible. Modern network requirements in terms of agility, security, management, etc., are not met by such fixed functionality devices. This led to the introduction of Software-Defined Networking (SDN) which has fully decoupled control and data plane [16]. Though SDN provides a flexible control plane through protocol such as OpenFlow [17], the data plane is still fixed in functionality. Let us consider a scenario where a researcher wants to develop their own transport protocol. But even with SDN, the data plane protocols are fixed, which means these devices will not be able to process the new transport protocol. For this, the researcher may have to build the custom ASIC and program them to support their protocol. This adds to

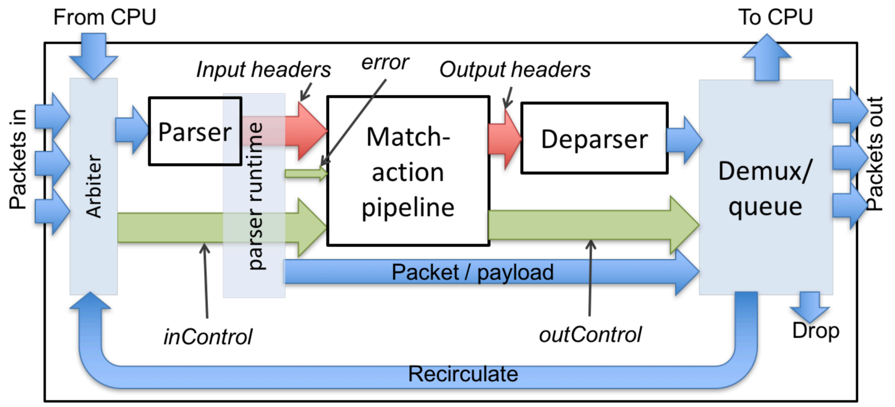

Figure 2.2: Very Simple Switch (VSS) architecture [1].

the cost and time required for such research. Programmable data plane addresses this problem and hence is an important stepping stone towards realizing next-generation networking, services, and applications [18].

Programmable data plane, as the name suggests, provides a way to express custom packet processing routines directly in the data plane [19]. This makes it possible to design and experiment with newer protocols and also provides an ability to create, add, and update the packet data in the device itself at a line rate. For instance, we can use programmable data plane devices to embed network telemetry data directly into the packets to create high-precision network monitoring that works at a line rate. Figure 2.2 represents a primitive Very Simple Switch (VSS) architecture. This architecture provides a high-level idea of how a programmable switch works. VSS receives the packet from one of its ports or from the port connected to the CPU directly. Then the packet goes through the parser stage which feeds the parsed packet into the programmable match-action pipeline. Finally the packets are reconstructed in deparser stage and queued for egress. The egress can send the packet through one of the output ports, recirculate it to the parser stage, send it to port directly connected to CPU or drop it. We further explore the use cases of data plane programming using INT in the related study section.

2.3 P4 Programming Language

Though it is possible to program the programmable data plane devices using low-level languages, having a specific programming language that properly reflects the internal architecture makes it much easier to program them. Protocol-Independent Packet Processors (P4) is currently the most widely adopted programming language for data plane devices [20]. It provides programming constructs that are specific to the underlying hardware making it easier to program the device data plane [21].

P4_16 is the current standard of the P4 programming language [22] known as Portable Switch Architecture (PSA). Figure 2.3 shows different stages of PSA at a high level. When the device first encounters a packet, it goes through the *parser* stage. In this stage, the packet data is parsed to extract required headers and payloads. At *ingress*, programmable match-action pipelines are used to provide generic packet processing capabilities, usually to make packet forwarding decisions. A Match-action pipeline is an abstraction where packets are matched against specific entries in the device table, and corresponding actions are executed. The tables required for this are populated and controlled via the control plane. Then the packet, along with its metadata, are buffered and optionally can be replicated to send to multiple egress ports at the *packet buffer and re-circulation* stage. Before the packets are emitted from the egress port, they go through an additional round of *egress* match-action pipeline. At this stage, addition/changes to the payloads can be performed, such as attaching telemetry information to the packets. Finally, the packet is deparsed at *deparser* stage and queued to exit from the specified egress port. Each packet goes through these stages whenever they are encountered by programmable data plane devices running P4 program during their traversal through the network.

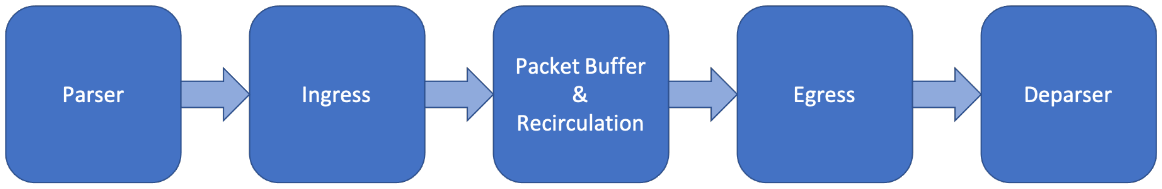

Figure 2.3: Stages of Portable Switch Architecture (PSA).

2.4 In-band Network Telemetry

Flow-level and port-level statistics collection using technologies such as SNMP, NetFlow, etc., have been a primary method of network monitoring. These techniques work well for a longer monitoring period to uncover network performance issues but fall short in detecting issues in a very short monitoring period. Due to such limitations, monitoring based on such techniques is not able to capture transient network events [23]. INT is a method collecting network telemetry information directly from the data plane. This technique enables monitoring of the network at the packet level. On top of that, the monitoring can be done at the line rate of the device, which means the monitoring performance can scale automatically as the line rate of the device increases. Hence, INT provides access to precise network information at a very high frequency addressing the limitations of existing monitoring methods. We discuss the details of using INT in chapter 3.

Bmv2 is a reference P4 software switch [24] written in C++11. It takes in JSON as an input which is generated by compiling P4 programs, and uses it to define packet processing behavior as specified by the P4 program such as INT. Bmv2 is not a production-grade switch and is mainly used for programmable data plane research purposes.

2.5 Literature Review

Task scheduling at the edge is one of the fields of active research. Authors have used various metrics such as network, cache, compute, energy, etc., in building efficient scheduling algorithms in their research. Also, the use of machine learning, heuristic-based approaches are widespread. Programmable data planes are relatively new though the use-cases and popularity have been growing. We believe that this is the first study that makes use of programmable data planes and INT to enhance task scheduling at the edge. We present various studies related to task scheduling in edge computing and INT below.

2.5.1 Task Scheduling in Edge Computing

Executing performance-critical tasks or tasks demanding additional compute resources than the edge device can provide requires the task to be offloaded to edge servers and compute nodes. The selection of sub-optimal edge servers for the tasks could increase communication costs, drastically increasing the task completion time. It might lead to poor task execution performance and might not meet the requirements for certain classes of applications running at the edge of the network, such as real-time applications. Various research has been done in task scheduling to improve the performance of edge computing. In order to find optimal task scheduling policy, Liu et al. [25] propose a two-step process. First, a decision is made whether to execute the task locally or to offload it to the Mobile-Edge Computing (MEC) server. Then a decision on where to schedule the computation tasks is made based on the queueing state of the task buffer, the execution state of the local processing unit, and the state of the transmission unit in order to achieve a shorter average execution delay. An adaptive neuro-fuzzy inference system is employed by Rashidi et al. to predict the availability of network and compute resources that are used to select offload targets optimally [26]. This resulted in higher resource utilization, better quality-of-service, and increased tolerance of network dynamics in task offloading. Chen et al. went a step forward and studied multi-user computation offloading problems in mobile-edge cloud computing environment [27]. They used the game-theoretic approach to formulate an efficient distributed computation offload algorithm that scales well as the user size increases. Energy efficiency is considered one of the vital parts of Mobile-Edge Computing (MEC). In addition to having lower computational resources, the devices usually have a limited power supply in such an environment. Zhu et al. have proposed algorithms for scheduling tasks with strict deadlines in an energy-efficient manner [28]. While edge computing brings computation closer to the edge devices, improving latency and reducing energy consumption, but inappropriate cache placement and utilization can degrade the overall system performance [29]. Li et al. have proposed a cache-aware task scheduling method to solve this issue by formulating a utility function that considers data chunk transmission cost, caching, and cache replacement penalty. Data used for tasks are placed in the optimal edge servers to maximize the utility value, thereby improving the performance. Even though the edge compute servers have better capabilities than the edge devices, the resources are still generally limited than that of cloud servers. Cooperation between edge servers and cloud servers is desired when additional resources are required to complete the tasks. Zhao et al. has proposed a cooperative scheduling scheme so that the tasks can take advantage of the low-latency of edge computing and also the abundant computational resources of cloud servers when required [30].

2.5.2 In-band Network Telemetry

In-band Network Telemetry (INT) can be used to perform network monitoring at the packet level. Since it can perform monitoring operations at the line rate, it becomes an excellent alternative to existing monitoring systems. At the same time can provide fine-granular metrics such as port-level queue usage statistics, accurate delay information, etc. Such precise link load information obtained by using INT is used by Yuliang et al. to develop High Precision Congestion Control (HPCC) that improves the flow completion times by up to 95% [31]. Programmable data planes on top of which INT work improves the agility of the network by allowing it to steer the traffic with great precision. These features are exploited by Lim et al. to improve network load balancing by quickly identifying the congestion events and steering the traffic through the less congested paths [32]. Pelle et al. have shown that the telemetry data obtained using INT can be used to optimally place the serverless functions improving the performance of latency-sensitive edge applications [33]. While INT can be used to detect various events, reducing the number of events to monitor can significantly reduce the overhead of the INT data collection. Such a programmable event detector capable of reporting only selected information is designed by Vestin et al. [34]. Similarly, Kim et al. have proposed to monitor a specific ratio of packets instead of all packets in order to reduce overhead while maintaining the detection of all significant data plane events [35]. In addition to the overhead of the data collection, storage and processing of a huge amount of data received from INT becomes a challenge. Scaling such a solution becomes a challenge when the amount of devices in the network increases. Basat et al. has proposed Probabilistic In-band Network Telemetry (PINT) that drastically reduces the number of INT fields to store in each packet [36]. PINT maintains sufficiently high accuracy of measurements though the data collection is reduced, thereby addressing the problem.

Chapter 3

Network-Aware Task Scheduler

In this chapter, we lay down the details of our proposed Network-Aware Task Scheduler. Figure 3.1 shows a high level architecture of our proposed system. The scheduler is a standalone node in the network that carries out the selection of nodes for the offloading devices. INT is used to monitor the network by collecting telemetry from all the data plane devices. All of the nodes in the network sends a probe packet to the scheduler periodically. This probe packet is used by the programmable data plane devices to inject INT payload when it traverses through the network. These packets are then collected by the scheduler and processed. When an edge device needs to offload some task, it sends a query requesting optimal edge nodes to schedule the tasks. Then the scheduler processes the network information received via INT to find out the best nodes for the querying device. The scheduler returns a ranked list containing the addresses of the best nodes. Finally, the querying node selects the nodes from the list to offload the tasks.

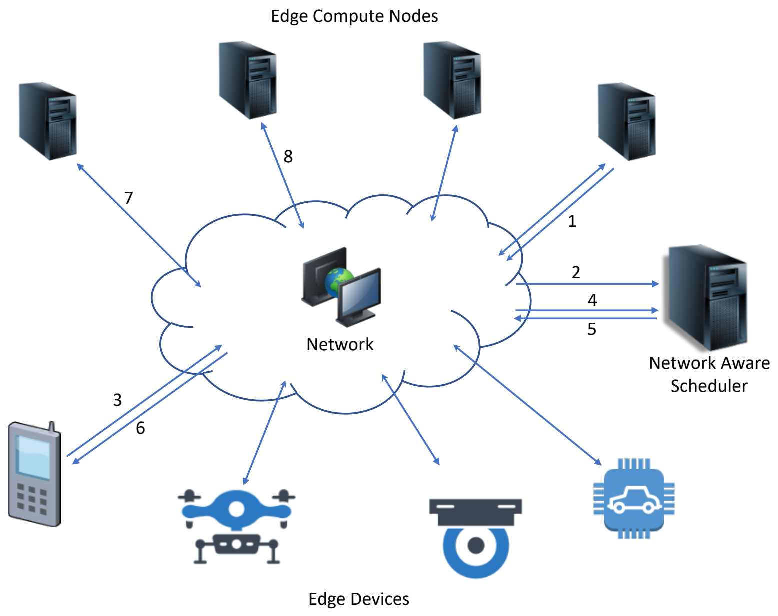

Figure 3.1: High level architecture of proposed network-aware task scheduler. (1) Probe packet collects the INT payloads from the devices. (2) The scheduler receives and stores the INT payload for further processing. (3,4) Edge device requests scheduler when it wants to schedule a task. (5,6) The scheduler responds with the list of best nodes with respect to the querying node. (7,8) Finally the querying device schedules the task.

3.1 System Design

Figure 3.2 shows the overall design of all the system components involved in the proposed network-aware task scheduler. Each edge node has three specific components: INT Probing, Task Offload Unit, and Task Execution Unit. *INT Probing* takes care of constructing the INT packets and sending them to the scheduler over the network. The *Task Offload Unit* is responsible for initiating optimal nodes query to the scheduler and performing the task offloading to the other nodes based on the received response from the scheduler. The *Task Execution Unit* receives the offloaded tasks, executes them, and sends back the results to the offloading node. Switch, or programmable network devices, has two specific modules: INT Probe Handler and Normal Traffic Handler. *INT Probe Handler* attaches the INT payload to the probe packet, which is then forwarded to the scheduler. All the other traffic, including ranking query from the offloading node, is handled by the *Normal Traffic Handler*. The scheduler has two specific tasks: receiving probe packets and handling the edge nodes' query requests. The *INT Probe Receiver* listens to all the probe packets and then sends the packet to *Network Mapping* for additional processing. *Query Server* is responsible for listening for the ranking query. It invokes the *Nodes Ranking* module receiving the appropriate nodes list and responds to the querying node. Both the *INT Probe Receiver* and *Query Server* uses UDP protocol in this system.

The Network-aware scheduler works in three distinct phases: INT Telemetry collection, Network Mapping, and Ranking of nodes.

3.2 INT Telemetry Collection

In order to create network awareness in the scheduler, we need to monitor the network and feed the data into the scheduler. In our case, the scheduler receives data from the devices in the form of INT payloads. In order to collect such metrics, there are two possible

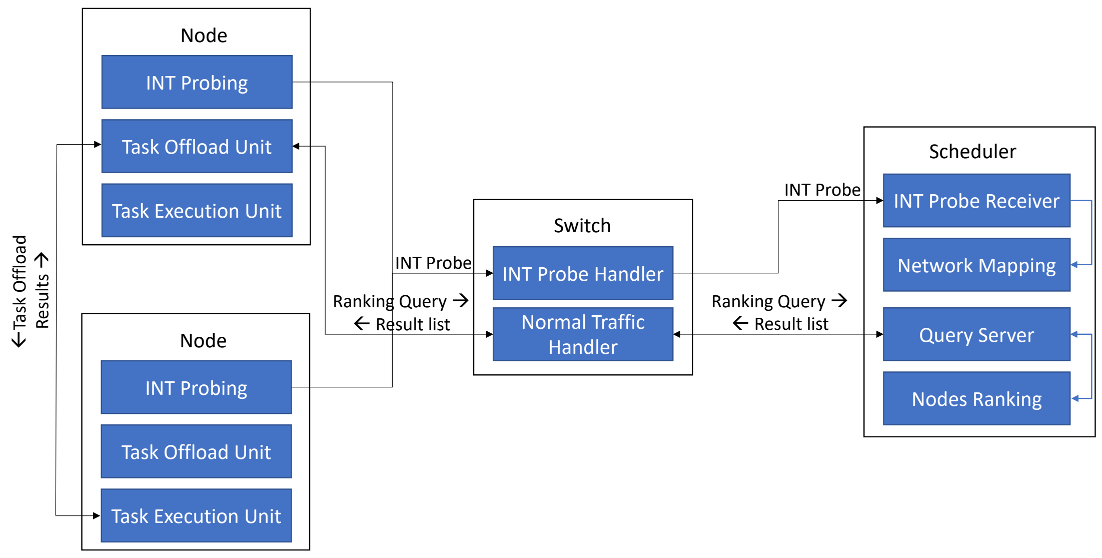

Figure 3.2: Overall system design of network-aware task scheduler.

approaches. One approach is to add the information to all the packets that traverse through the network. But there are drawbacks of this approach. Adding additional information to each packet will increase the amount of traffic. Research shows that adding just two fields of INT on every packet demands 4.2% of the entire packet payload in a network of just five switches [31]. When the size of the network or the amount of INT data in each packet is increased, this figure becomes much higher, which is entirely undesirable. Also, there is a limit on the maximum size of a packet allowed in a network. Adding fields to every packet might increase the size of some packets beyond the allowable limit, which can cause issues.

Due to the aforementioned reasons, we are using dedicated probe packets scheduled at specific intervals in our approach. INT fields are attached to these probe packets rather than adding it to all of the packets. We use the device's internal memory to store desired metrics using the P4 program. This P4 program calculates and stores information in switch registers as each packet passes through the device by using ingress and egress match-action pipeline [37]. Given that the information stored in registers is rather updated, not added

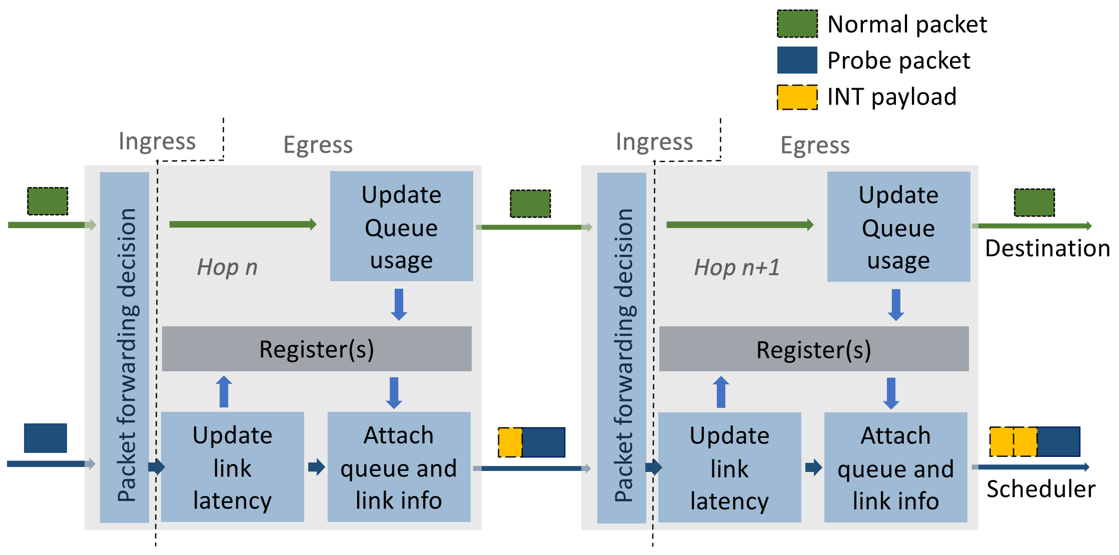

Figure 3.3: P4 based data plane processing pipeline of probe packets.

each time the device/switch encounters a new packet, the memory requirement is constant. When the switch encounters the probe packet, the same P4 program attaches the data stored in the memory to its payload and forwards it to the next hop. This way, a considerable amount of data is reduced by adding the INT fields to the probe packet only.

Now that we have established the importance of using probe packets, we discuss the details of the probe packet(s) in action. The first thing is that each network device running our P4 program should be able to identify the probe packet. For this, we use UDP packet with certain IP class field set referred to as Geneve option [38]. Our P4 program parses the incoming packet and looks for that specific field. A packet with the Geneve option field set to a specific value is considered the probe packet. Figure 3.3 shows the overall overview of how probe packets are processed at each hop. We can categorize all the network packets into normal and probe packets. When the network device receives normal packets, the P4 program extracts information such as queue occupancy, ingress/egress timestamp and updates the device's appropriate register(s). Then the packets are forwarded without any modifications. Although the normal packets are part of the metric collection process, they do not carry any collected monitoring data directly. Now when the devices receive probe packets, all the collected telemetry information is added to the packet as a INT payload and then is forwarded to the next hop. Then the values in registers are reset. Apart from collecting the payload, the probe packet is crucial in calculating the link latency information. The previous device that encountered the probe packet attaches an egress timestamp to it. The following device then extracts it at its ingress stage, where the timestamp is extracted and then compared with the current timestamp to determine the link latency. This information is used to capture the state of jitter in any link, although the link latency is relatively stable over time. Such information can be crucial to inferring the overall status of the network being monitored. Please note that we have synchronized our switch instances using Network Time Protocol (NTP) to enable such kind of measurement. Pseudocode 1 and 2 present the details of ingress and egress of the P4 program, respectively.

Algorithm 1: Pseudocode of P4 program (Ingress) running on all network devices.

```
1  $SW\_ID$: ID of the device P4 program is running on;
2  $PKT$: Packet currently being processed;
3  $META$: Metadata of the current device and packet;
4  $T\_FWD$: Match-Action table with forwarding port information;
5  $A\_DROP()$: Action to drop the packet;
6  Function Ingress():
7      if $PKT[destination\_addr]$ in $T\_FWD$ then
8          $destination\_port = T\_FWD[PKT[destination\_addr]][port]$
9          $META[dst\_port] = destination\_port$
10     else
11         $A\_DROP()$
12     end
13 end
```

As we rely upon the probe packets for network monitoring, we must capture the information from all the available paths so that the scheduler has up-to-date information about the whole network. For this purpose, we schedule probe packets from all the available devices in the network. Though this approach might result in some probes traveling the

Algorithm 2: Pseudocode of P4 program (Egress) running on all network devices.

```
$SW\_ID$: ID of the device P4 program is running on;
$PKT$: Packet currently being processed;
$META$: Metadata of the current device and packet;
$R\_max\_queue[N]$: Register to record max queue length of N ports of the device;
$R\_link\_lat[M]$: Register to record link latency of M neighbours of the device;
$Fn\_is\_probe(pkt)$: Check if the packet is a probe;
$Fn\_extract\_INT(pkt)$: Extract INT payload from the packet;
$Fn\_attach\_INT(pkt, int)$: Attach INT payload to the payload;
$A\_FWD(pkt, port)$: Action to forward the packet to the given port;
Function Egress():
    $destination\_port = META[dst\_port]$
    $last\_queue\_len = R\_max\_queue[destination\_port]$
    $current\_queue\_len = META[port\_stats][destination\_port]$
    if $current\_queue\_len > last\_queue\_len$ then
        $R\_max\_queue[destination\_port] = current\_queue\_len$
    end
    if $Fn\_is\_probe(PKT)$ then
        $INT = Fn\_extract\_INT(PKT)$
        $(from\_swid, ts) = INT[link\_info]$
        $current\_timestamp = META[egress\_timestamp]$
        $R\_link\_lat[from\_swid] = (current\_timestamp - ts)$
        $INT[link\_info] = (SW\_ID, current\_timestamp)$
        foreach $sw\_id$ in $R\_link\_lat$ do
            $latency = R\_link\_lat[sw\_id]$
            $INT[SW\_ID][latency].append((sw\_id, latency))$
        end
        foreach $port$ in $R\_max\_queue$ do
            $max\_queue = R\_max\_queue[port]$
            $INT[SW\_ID][queue].append((port, max\_queue))$
        end
        $Fn\_attach\_INT(PKT, INT)$
        $Reset()$
    end
    $A\_FWD(PKT, destination\_port)$
end
Function Reset():
    foreach $port$ in $R\_max\_queue$ do
        $R\_max\_queue[port] = 0$
    end
end
```

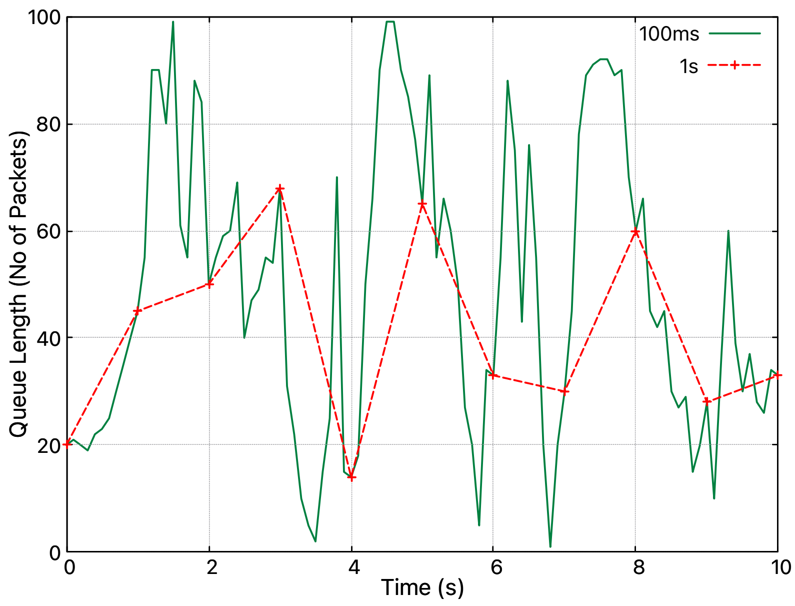

Figure 3.4: Comparison of sampling of egress port for queue length at $100ms$ and $1s$.

same section of the network more than once, or some paths might even be left out, we leave the optimization of the probing path to future work. Apart from path selection, probing frequency is another important criteria to consider. In this study, we are focusing on high-frequency network monitoring. Figure 3.4 compares the queue length sampled from egress port at two different rate: $100ms$ and $1s$. As we can see from the figure, having a lower sampling period captures subtle changes in the network. Having a higher sampling period means greater loss of network information. That means SNMP which has a sampling period of tens of seconds, has a lower chance of capturing subtle network statistics. Hence we have set our probing interval to 100ms. Due to our testbed's limitations, we could not reduce the probing interval further than this in the current study. Hence, we have 10 probe packets per second traversing through any specific path at any given time (considering the path optimizations) which results in a considerable reduction in the network overhead compared to the scheme where INT payload is added to every packet traversing the network.

3.3 Network Mapping

The purpose of the network-aware scheduler is to serve queries from the devices and provide them with a list of the best available nodes. In order to be able to do that, the scheduler needs to learn the overall network topology and status of each device in the network. This is another purpose that is served by probing. When the device or nodes schedule a probe packet, the INT payload is attached to them in the order they encounter the network devices. Let us consider a probe packet has INT payload from device $D1$, $D5$, $D6$ in order. With that information, we can confidently say that devices $D1$ and $D5$ are connected, and so are devices $D5$ and $D6$. Also, the devices $D1$ and $D6$ are connected via $D5$. This information is represented in the scheduler as a graph of connected nodes. Each node in the graph represents specific devices, and the information it holds corresponds to the INT data received from the probe. Every probe received by the scheduler updates the appropriate nodes in this graph.

The scheduler runs a graph traversal between the querying node and all the available edge servers to serve the query from edge devices. This way, the scheduler tracks down all the devices and paths between edge devices and the edge servers. Let us consider an example where we have three edge devices and nodes, namely $E1$, $E2$, and $E3$. When the scheduler receives the query from $E1$, there are two possible offload points: $E2$ and $E3$. Then scheduler runs graph traversal to find out the network devices and path between $E1$, $E2$ and $E1$, $E3$ respectively. Now the scheduler quantifies the network state between those two possibilities and ranks the list of results with it. Finally, the results are returned to the querying edge $E1$ device. The device will then make the edge server selection based on the results.

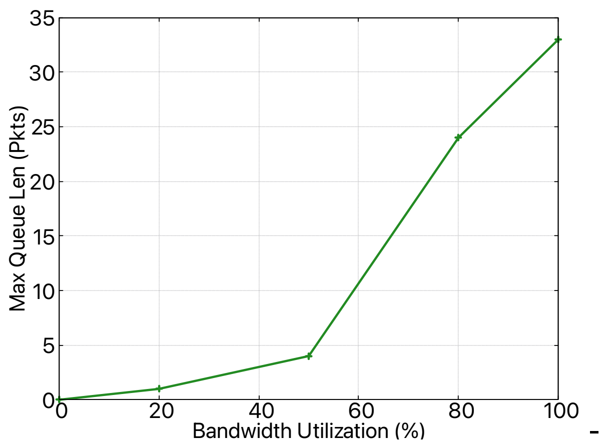

Figure 3.5: Max queue length of egress queue at different bandwidth utilization levels.

3.4 Methods of Ranking the Nodes

It is clear from above that the scheduler needs a method to rank the available nodes based on the network conditions with respect to the querying node. In our study, we present two ranking methods. In the first method, available nodes are ranked based on delay with respect to the querying node, and in the second method, bandwidth is used as a metric for ranking the available nodes. This means the selection can be made based on the lowest delay between nodes or the highest bandwidth availability between the nodes. In another approach, the scheduler can optionally return a list of all the edge nodes along with their bandwidth availability and delay from the querying node so that the querying node can make its own decisions. In this study, we focus on the scheduler making decision based on the delay and bandwidth availability and leave the latter approach for future work.

3.4.1 Delay-based Node Ranking

A packet can experience two different kinds of delay when traversing through the network. It can experience a delay in the link and in the device/hop itself. For formally defining the delay that we are considering in this case, let us consider a network consisting of a set of edge devices and servers E, a set of network devices D. A set of links

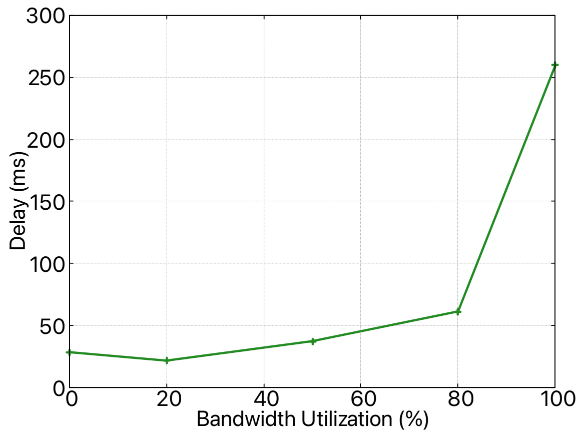

Figure 3.6: Delay experienced by the packets at each hop at different bandwidth utilization levels.

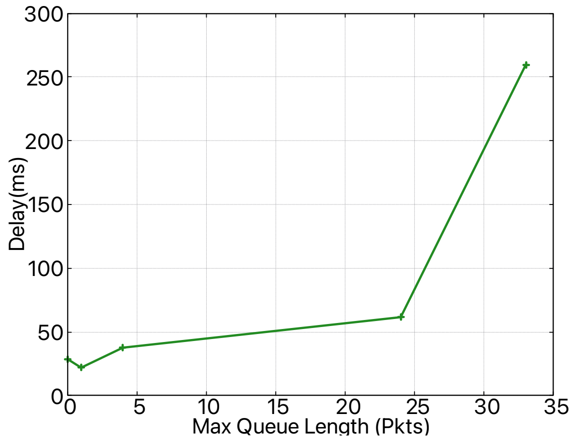

Figure 3.7: Inference from figure 3.5 and 3.6 that delay increases as the queue occupancy increases.

L connects all the edge devices, servers, and network devices. Let us consider two edge nodes E_n and E_m where $E_n, E_m \in E$ are communicating, and we want to calculate the delay between these devices. Communication packets between the edge nodes traverses through arbitrary number of network devices and switches $H_1, H_2, \ldots H_k \in D$ and links $L_1, L_2, \ldots L_k \in L$. Then, we can calculate one-way delay between the communicating nodes as $Delay(E_n, E_m) = \sum_{i=1}^{k} delay(L_i) + \sum_{i=1}^{k} delay(H_i)$. We are already calculating the link delay L_i using the probe packets as discussed in section 3.2. We are interested in calculating the hop latency that completes the total latency calculation required for a delay-based ranking scheme.

In order to infer the amount of delay each packet experiences at each hop, we use queue occupancy information received from the probe packets. Typically, increased queue occupancy in any device would mean that the packet would potentially experience more delay when going through that device. Our study found that minimum queue and average queue occupancy information is inconclusive to infer the delay experienced by the packets. The reason is that the occupancy is usually zero, which means that minimum queue occupancy would be zero during most of the measurement intervals. Hence, average queue occupancy is also pulled towards zero. Therefore, we rely on maximum queue occupancy for the trend inference. Figure 3.5 and 3.6 gives us information about how the queue occupancy indicates the delay experienced by the packets in a network device when the bandwidth utilization is increased. For this experiment, we used Behavioral Model (Bmv2) switch, which is an experimental reference software switch that supports P4 in Mininet. Please note that Bmv2 is an experimental switch and has performance limitations which is not the case for the production-grade switches. For this reason, we define our maximum bandwidth to be 20 Mbps, 10ms link delay in a Mininet network of two hosts connected by Bmv2 switch. Fixed-rate *iperf* transfers were executed between two hosts. That is, 50% bandwidth utilization would correspond to setting *iperf* transfer to 10 Mbps. The Bmv2 switch is running our P4 program that stores the maximum queue occupancy between the probing interval to the switch registers. We set the probing interval to 100ms. The values in the registers are reset after encountering the probe packet to enable registers to track maximum queue occupancy of another 100ms interval. During the same time, we run the ping tool to calculate the end-to-end delay between the hosts. We run the transfers mentioned above for 300 seconds for each bandwidth utilization and record the maximum queue occupancy and average ping-based delay for that period.

From the figure 3.5 and 3.6, we can see that the maximum observed queue occupancy and delay experienced by the packets increases significantly as the bandwidth occupancy is increased. Figure 3.7 shows the inferred relation between delay experienced by each packet at egress at various queue occupancy levels. Also, we can see that packets in the queue increase sharply when the bandwidth utilization exceeds 50%. We can see a similar trend in end-to-end delay. The delay stays between $40 - 60$ms which is closer to the normal RTT of about 40ms up until the bandwidth utilization reaches 80%. A sharp increase (about $6x$) can be seen as delay reaches 250ms when the bandwidth utilization exceeds 80%. This shows a positive correlation between the maximum queue occupancy and the delay experienced by the packet. We exploit this correlation to quantify delay from the maximum queue occupancy observed from the INT data. We introduce a conversion factor k to linearly convert the queue occupancy to delay observed. Our experimental evaluation showed that the value of $k = 20$ms performs better in inferring the delay experienced by the packets in the network device in our topology resulting in better performance of the network-aware scheduler overall. More work can be done in the future in regards to fine-tuning this parameter. Algorithm 3 shows the overall ranking process using the delay between the edge servers and the querying node. First, the network graph representation created by the scheduler is queried to get all the edge nodes and servers reachable via the querying node upon receiving the query. The method, as mentioned earlier, is used to calculate the overall delay by summing up the link delay and the hop delay at each link and network device. The list of the nodes is then sorted using the delay factor so calculated before sending the results to the querying node.

Algorithm 3: Sort the edge servers based on the overall network delay with respect to the querying 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
        $totalLinkDelay += D(l_i)$;
    end
    $totalHopDelay = 0$ ;
    foreach $h_i$ in $H(e_n, e_i)$ do
        $totalHopDelay += k * Q(h_i)$;
    end
    $\Delta = totalLinkDelay + totalHopDelay$;
    $N.append((e_i, \Delta))$
end
$N = S(N)$ ;
return $N$;
```

3.4.2 Bandwidth-based Node Ranking

As seen from the figure 3.5, there is a positive correlation between the maximum queue occupancy and bandwidth usage. We use this correlation to formulate a bandwidth-based node ranking scheme. To make it easier to formulate, let us consider two nodes where

$E_n, E_m \in E$ are communicating, and we are interested in inferring available bandwidth between them. The packet traverses through links $L_1, L_2 \dots L_k \in L$ and network devices $D_1, D_2 \dots D_k$ which has available bandwidth $B_1, B_2 \dots B_k$ respectively. Its easy to see that the available bandwidth between E_n and E_m is the minimum available bandwidth in the link represented by $throughput(E_n, E_m) = min(\{B_1, B_2 \dots B_k\})$. This is because the bottleneck link bandwidth is the maximum bandwidth achievable in data transfers between any devices. This means available bandwidth $B_1, B_2 \dots B_k$ is negatively correlated (positively correlated with usage) with the queue occupancy $Q_1, Q_2 \dots Q_k$. Hence, we can say that the network device with maximum queue occupancy is the bottleneck that defines the overall communication bandwidth between communicating edge devices. Then the scheduler uses this information to sort the nodes based on the available bandwidth and sends the response to the querying node similar to the algorithm 3.

Chapter 4

Experimental Analysis and Results

In this section, we provide the details on the experiments carried out in this study and discuss the obtained results. Mininet is used to emulate the experimental network. Mininet is a network emulator capable of emulating large networks used usually for research purposes [39]. The main goal of Mininet is to ease the SDN research by providing tools to emulate a large network in a single computer for rapid prototyping. It uses Linux network namespace to provide process-based virtualization to create multiple hosts on a single machine. The current version of Mininet also has experimental support for distributing the network topology among multiple clusters for significantly larger experiments [40].

Each virtual host in the network runs probing, and experimental compute offloading application services. Behavioral Model (Bmv2) is that standard reference P4 software switch used which provides connectivity among all the virtual hosts in the network. All instances of the switch are running the P4 program required for this experiment. Due to the multiple instances of hosts, switches, and multiple services used per virtual hosts instance, we need to scale out the experiments for better performance. For this, we used the cluster support in Mininet to scale our experiments to multiple physical servers. Figure 4.1 shows this topology with 8 nodes and 12 Bmv2 switches used in our experiment. This setup runs on

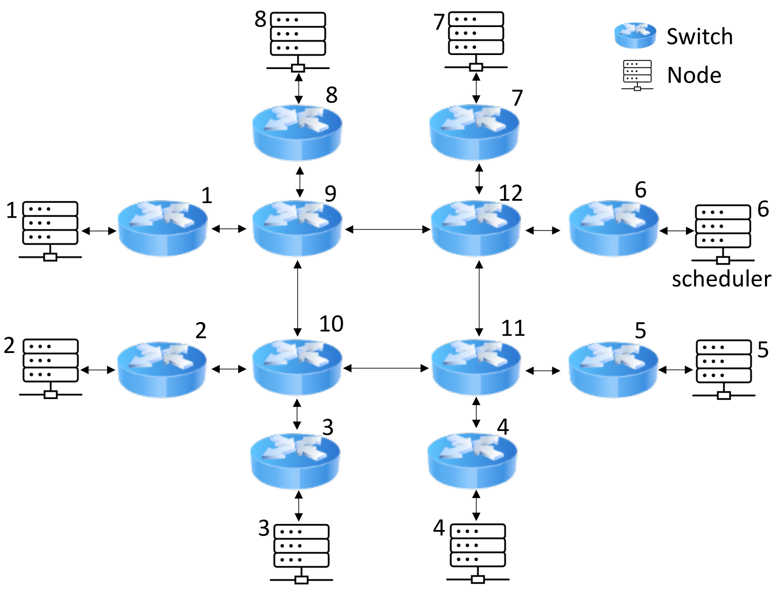

Figure 4.1: Experimental topology realised in Mininet. Node (6) acts as network-aware scheduler. Other nodes acts as edge devices or servers. Edge devices offloads tasks to the edge servers and edge servers execute the tasks.

four physical servers, each with four core CPUs and 32 GB RAM with Ubuntu 18.04 as the Operating System. We also use HP Procurve switches to provide physical connectivity between the servers.

As shown in figure 4.1, node 6 acts as the network-aware scheduler. It can both receive probe packets and also serve ranking queries from the other nodes. We have two additional services running in all of the nodes. The probing service is configured to send probe packets to the scheduler at specific intervals from all nodes except the scheduler itself. Another experimental compute offloading module is installed on all the nodes. At any given time, nodes can act as edge devices that can request offloading or edge servers that can execute the offloaded tasks. All of the mentioned services are implemented in the Python programming language. All of the switches in the topology are running our P4 program to handle the probe packet as shown in pseudocode 1 and 2.

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 4.1: Data size, execution time for different workload sizes used in the experiments.

In order to evaluate the performance of the proposed INT-based task scheduler, we compare it with two different scheduling strategies: nearest node(s) scheduling and random node(s) scheduling. In the nearest nodes(s) scheduling strategy, the edge devices always schedule the tasks to the nearest nodes only in the pursuit of achieving low-latency communication. Since all the links have a 10ms delay, we can see that the nodes that are minimum hops away are the closest ones to each other. For example, nodes 3 and 4 are the closest ones to each other in figure 4.1. Since we assume that we know the network topology beforehand, we calculate the nearest nodes ahead of time for simplicity. In the random nodes(s) scheduling strategy, as the name suggests, the nodes are selected at random by the offloading device as task offload targets. This strategy simulates the random load balancing of the tasks compared to the nearest nodes(s) scheduling strategy.

We used two different workload types for our experiments that represent most of the workloads in the real-world task offloading scenarios at the edge: serverless computing workload and distributed computing workload. Serverless computing is a paradigm where execution happens in the cloud rather than in the local environment [41]. This allows the developers to focus on their business logic of the function and offshore the low-level management details such as running the function, scaling, optimizing, etc., to the serverless computing provider. Function-as-a-Service is a type of serverless computing in which the computation unit is functions. All major cloud provider provides serverless computing platform such as AWS Lambda [42], Azure Functions [43], and Google Cloud Functions [44].

In the serverless computing workload, the offloading node selects only one offloading node to offload its tasks. This is representative of function-as-a-service type workloads where a significant amount of time is spent on the network communication providing significant potential for optimization using edge computing. In this type of computing, the edge devices execute remote functions residing on the edge server to perform specific tasks that cannot be done with the resources on board [15].

On the other hand, distributed computing workload represents all the tasks that take more than one compute node or server to complete. The distributed system provides critical advantages such as scalability, fault tolerance, performance, etc., which are required as the demand for processing power and data storage increases. We can find abundant applications that use distributed computing paradigm, such as federated machine learning applications. Low-latency communications potentially boost the performance of the applications such as deep learning in a distributed environment, which makes edge computing a viable candidate [45]. For our experiment, we configured the compute offloading module to submit one task to a single offloading target for serverless computing and selected three different targets to offload three different tasks for the distributed computing workload. Table 4.1 shows the different workload sizes used for the experiment. Each workload size is defined by the amount of data transfer it makes and the amount of task execution time on the offloading target is expected. Finally, we use *iperf* tool to create background traffic and congestion scenarios in the network. We run one or two background data transfers between randomly selected nodes for the duration of 30 or 60 seconds periodically. Since we are comparing our method of selecting the offload targets with the other two baseline methods, we ensure that all the methods are subjected to the same background traffic conditions to ensure consistency in the experiment. We run 200 task offloading per experiment and categorize our results according to the workload types and sizes under each node ranking strategy. Please note that the results presented here are the part of the research published

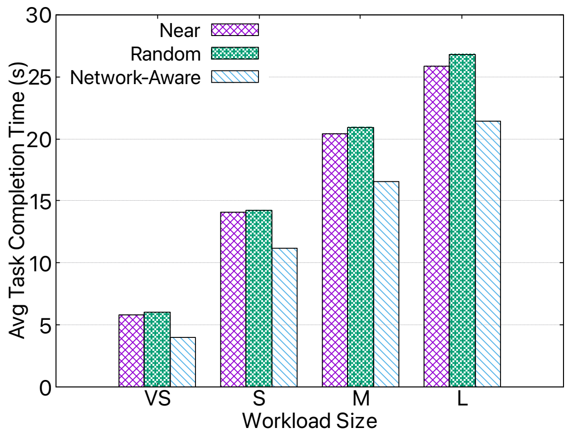

Figure 4.2: Average task completion time (s) of *serverless computing workload* using *delay-based* node ranking strategy for network-aware task scheduling compared to the near and random node selection based scheduling strategy on various workload sizes: very small (VS), small (S), medium (M) and large (L).

for this study [9].

4.1 Delay-based Node Ranking

Figure 4.2 shows the performance comparison of network-aware task scheduler using the delay-based ranking strategy with near and random scheduling strategy on workloads of various sizes. Figure 4.3 shows the corresponding performance gain of our method compared to the nearest nodes selection-based scheduling strategy. We observe that our network-aware scheduling strategy using delay-based nodes ranking yields performance between $17-31\%$ on the various workload sizes. Results show that the performance gain is highest for the workloads of the smallest size. This means that this strategy is well suited for smaller workloads with low total data transfer and execution time. Similarly, figure 4.4 and 4.5 show a comparative analysis of average task completion time and performance comparison for the distributed computing workloads, respectively. The network-aware scheduling outperforms nearest-node and random nodes selection-based scheduling strategies regardless of the workload sizes. The performance gain of network-aware strategy over other strategies

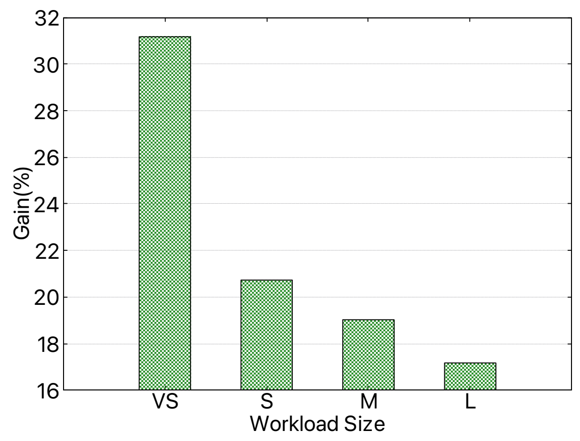

Figure 4.3: Performance comparison between network-aware scheduling using *delay-based* nodes ranking strategy and nearest node selection-based scheduling strategy for various workload sizes: very small (VS), small (S), medium (M) and large (L) on *serverless computing workload*.

ranges between $7 - 13\%$, and still, the smaller workloads perform better compared to other workload sizes. This is because smaller workloads have shorter total task completion time and are better poised to gain from network-aware strategy. If we look at the larger workloads, the network conditions might worsen after the task offloading has started. Then the task's performance will suffer for a longer time impacting overall task completion time. Whereas in the case of smaller workloads, such impact is short-lived.

4.2 Bandwidth-based Node Ranking

Since our strategy is about reducing the communication overhead by favoring offload targets that are optimally positioned with respect to the offloading device, we present average data transfer time comparison between network-aware scheduling strategy using bandwidth-based ranking strategy and nearest nodes, random nodes selection-based scheduling strategy for different workload sizes in figure 4.6. The result shows that the network-aware method performs better than both the near and random nodes selection-based strategies. Figure 4.7 shows that the performance gain from network-aware strategy ranges between $28 - 40\%$

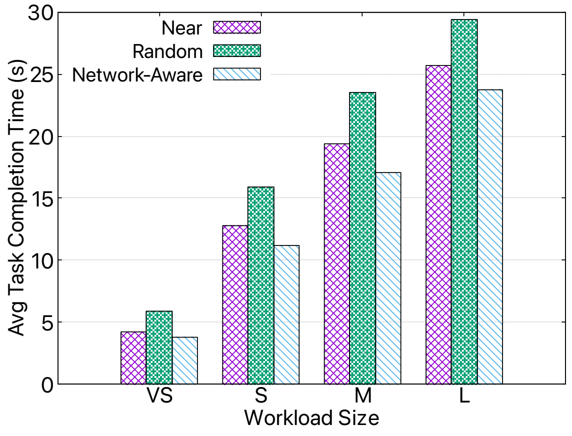

Figure 4.4: Average task completion time (s) of *distributed computing workload* using *delay-based* node ranking strategy for network-aware task scheduling compared to the near and random node selection based scheduling strategy on various workload sizes: very small (VS), small (S), medium (M) and large (L).

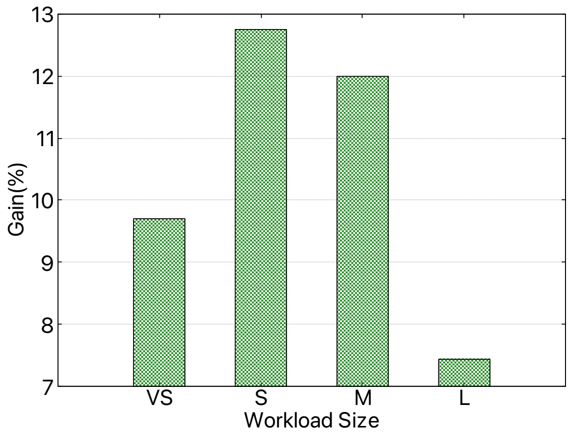

Figure 4.5: Performance comparison between network-aware scheduling using *delay-based* nodes ranking strategy and nearest node selection-based scheduling strategy for various workload sizes: very small (VS), small (S), medium (M) and large (L) on *distributed computing workload*.

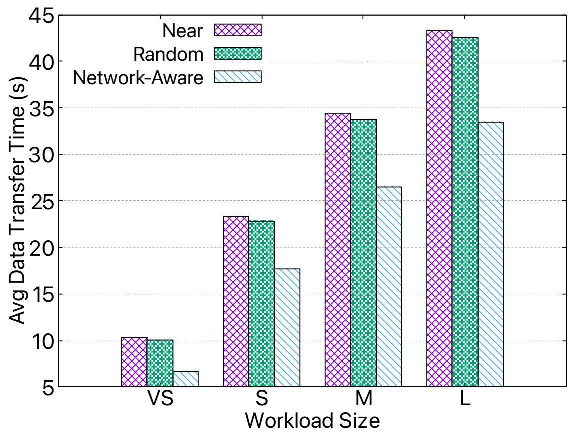

Figure 4.6: Average data transfer time (s) of *distributed computing workload* using *bandwidth-based* node ranking strategy for network-aware task scheduling compared to the near and random node selection based scheduling strategy on various workload sizes: very small (VS), small (S), medium (M) and large (L).

when compared against the nearest node selection strategy in terms of average data transfer time. If we account for the average execution time of each workload as presented in table 4.1, the performance gain ranges between $22 - 35\%$ in terms of average task completion time. Even in this strategy, the smaller workloads have better performance gains out of all workload sizes. We attribute this behavior to the same reason smaller workloads perform better in network-aware scheduling using delay-based ranking strategy in 4.1. For example, let us take three different transfers $T0$, $T1$, and $T2$ as shown in the figure 4.8 where available bandwidth between the nodes is decreasing over time. All transfer starts at $t0$ but ends at a different time $t1$, $t2$, and $t3$ respectively. $T0$ is short-lived, followed by $T1$ and $T2$, respectively. We can see that decreasing bandwidth availability between the nodes over time affects the transfer with the longer transfer time as those transfers cannot be rescheduled to a different node once it starts. This results in the degraded performance for such transfers. Use of SDN to change the paths of such transfer on the fly using a control plane is a possible approach to solve this issue. We leave the study and implementation of such optimization for future work.

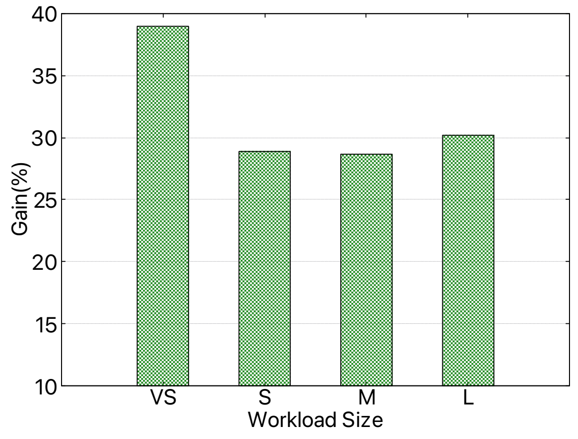

Figure 4.7: Performance comparison between network-aware scheduling using *bandwidth-based* nodes ranking strategy and nearest node selection-based scheduling strategy for various workload sizes: very small (VS), small (S), medium (M) and large (L) on *distributed computing workload*.

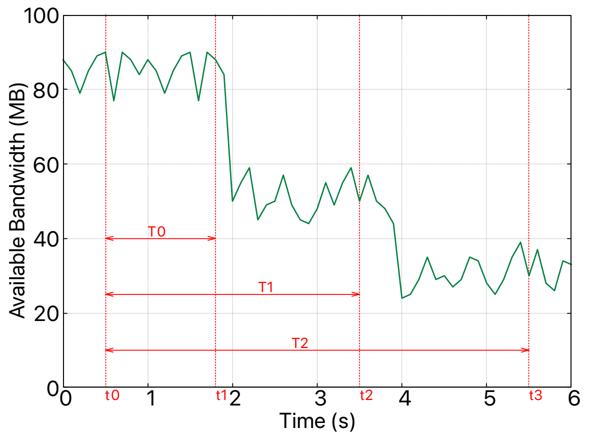

Figure 4.8: Comparison between three transfers with increasing transfer duration when the bandwidth availability is decreasing over time.

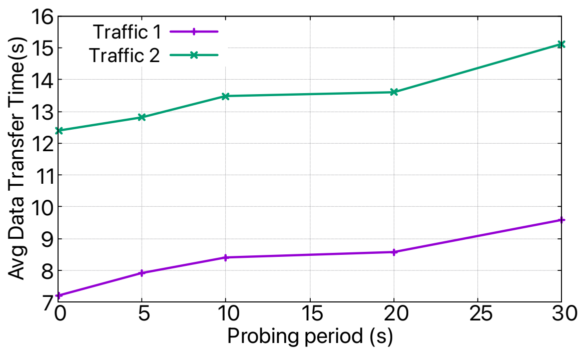

Figure 4.9: Impact of probing frequency of average data transfer time of the scheduled tasks under different traffic conditions.

4.3 Impact of Probing Frequency

We are using probe packets to extract INT metrics from every network device that helps make the scheduler network-aware. We can run probing at specific intervals. In this section, we evaluate the impact of probing frequency on the performance of network-aware task scheduling, specifically task completion time. Changing the probing frequency means increasing or decreasing the time between the probing, which would change how frequently the network statistics are updated in the scheduler that receives the probe packets. This has nothing to do with the actual working of the probe packets themselves. Our rationale for using programmable data planes and INT is that we can make network monitoring fast and precise enough to detect the subtle changes and congestion events. Hence, we hypothesize that a higher frequency of probing should increase the chances of the scheduler to detect those subtle changes that might lead to better performance.

For this evaluation, we choose probing intervals of 0.1s, 5s, 10s, 20, and 30s where 30s is in the range of typical SNMP polling interval. Distributed workload using bandwidth-based ranking strategy is used under two different traffic scenarios: *Traffic 1* and *Traffic 2*. *Traffic 1* uses a medium-sized workload and has background traffic that changes less frequently. For this, we run three *iperf* transfers between randomly selected nodes at the distance of

10 seconds from each other. Each of the transfers runs for 30 seconds which is followed by a 30 second sleep. *Traffic 2* uses a small workload size and has background traffic that changes much more frequently than *Traffic 1*. This is to simulate the more dynamic nature of the network. Same as before, we run three different *iperf* transfers between randomly selected nodes started at the gap of 10 seconds from each other with 5 seconds of transfer duration followed by 5 seconds of sleep.

Figure 4.9 shows the results of this experiment. A lower probing period or higher probing frequency seems to lower the average data transfer time resulting in higher performance gains on both *Traffic 1* and *Traffic 2* scenarios. Analyzing the data from the figure, we can see that the transfer takes 12.5 seconds to complete under *Traffic 2* conditions when the probing is 0.1ms. The same transfer takes about 15 seconds to complete when the probing period is 30 seconds which is about 20% increase. It indicates that having a high probing frequency increases the likelihood of capturing subtle changes in the network, enabling the scheduler to make more optimal choices of the targets. This result further supports our rationale of using high-frequency INT-based network monitoring over SNMP or NetFlow to create a network-aware scheduler. Due to the limitations observed in our distributed Mininet testbed, we capped our minimum probing interval to 0.1s. Hence, we leave minimum probing period optimization to the future work.

Chapter 5

Conclusion and Future Work

In-band Network Telemetry (INT) provides a mechanism of obtaining high-precision network telemetry directly from the data plane at the line rate. The information obtained is crucial to detect subtle changes in the network, which increases the ability to detect network congestion events more precisely with high accuracy. It increases the network visibility, which is not possible using the traditional form of network monitoring such as SNMP, NetFlow, etc. In this study, we proposed a network-aware task scheduler for edge computing that leverages INT. The scheduler uses the data obtained from the INT to infer the network conditions in order to optimally select the edge servers for task offloading such that overall task completion time is decreased, improving performance. We proposed a scheme where the production traffic is used only to update the telemetry data within the network device but is not affected at all during the INT data collection process. Instead, we use specially crafted probe packets to periodically collect the INT data from each network device with programmable data plane capabilities. The scheduler was subjected to various experiments running server selection strategies using our proposed methods of node ranking under different workloads. We observed that compared to the baseline strategies, network-aware strategy leads up to a 40% reduction in average data transfer times and up to 30% reduction in average task completion times. This shows that increasing the network visibility with INT for task scheduling at the edge yields promising performance improvements.

The current version of the scheduler only considers the network conditions for making the server selection decisions. In the future, we will enhance it with compute availability to ensure that load is balanced evenly among available resources. Also, the scheduler is currently centralized. This can be a possible source of a bottleneck or single point of failure. In the future, we plan to adapt a distributed scheduler approach by investigating the possibility of using the network devices to store all the required information so that the edge devices can omit communicating with the centralized scheduler. Considering the heterogeneous edge server environment is another promising direction for future work where tasks to be offloaded may have certain hardware (e.g., GPU) or software requirements that need to be considered while scheduling the tasks. Scheduler in the future can make use of availability of such requirements to properly schedule the tasks. Finally, we will investigate the optimization of transfers between edge devices and edge servers since previous studies show that utilizing network bandwidth can be challenging for high bandwidth networks [46–48].

Bibliography

[1] "P416 language specification," Jun 2020. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.1.html

[2] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel *et al.*, "Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[4] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 2020, pp. 1–6.

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

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

[7] C. Hare, "Simple network management protocol (snmp)." 2011.

[8] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes, "Cisco systems netflow services export version 9," 2004.

[9] B. Shreshta, R. Cziva, and E. Arslan, “Int based network-aware task scheduling for edge computing,” in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 879–886.

[10] S. Sinha, “State of iot 2021: Number of connected iot devices growing 912.3 billion globally, cellular iot now surpassing 2 billion,” Sept 2021. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[11] J. M. Khurpade, D. Rao, and P. D. Sanghavi, “A survey on iot and 5g network,” in *2018 International conference on smart city and emerging technology (ICSCET)*. IEEE, 2018, pp. 1–3.

[12] “Azure network round-trip latency statistics,” June 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/networking/azure-network-latency

[13] 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.

[14] 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.

[15] 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.

[16] K. Kirkpatrick, “Software-defined networking,” *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[18] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.

[19] 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.

[20] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *arXiv preprint arXiv:2101.10632*, 2021.

[21] 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.

[22] "P416 portable switch architecture (psa)," Apr 2021. [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[23] 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.

[24] "Behavioral model (bmv2)," https://github.com/p4lang/behavioral-model, 2021, accessed on 2021-10-10.

[25] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *2016 IEEE international symposium on information theory (ISIT)*. IEEE, 2016, pp. 1451–1455.

[26] 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.

[27] 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.

[28] T. Zhu, T. Shi, J. Li, Z. Cai, and X. Zhou, "Task scheduling in deadline-aware mobile edge computing systems," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4854–4866, 2018.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] B. Shrestha, “sbibek/int-edge,” 2021. [Online]. Available: github.com/sbibek/INT-Edge

[38] M. Hira and N. Katta, “In□band network telemetry (int),” https://nkatta.github.io/papers/int-hula.pdf, 2015, accessed on 2021-10-10.

[39] B. Lantz and B. O’Connor, “A mininet-based virtual testbed for distributed sdn development,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 365–366, 2015.

[40] B. Lantz, “Mininet,” https://github.com/mininet/mininet, 2021, accessed on 2021-10-10.

[41] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “The rise of serverless computing,” *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[42] Amazon, “Aws lambda,” https://aws.amazon.com/lambda, accessed on 2021-10-19.

[43] N. Mashkowski, “Introducing azure functions,” https://azure.microsoft.com/en-us/blog/introducing-azure-functions, 2016, accessed on 2021-10-19.

[44] Google, “Cloud functions,” https://cloud.google.com/functions, accessed on 2021-10-19.

[45] J. Chen and X. Ran, “Deep learning with edge computing: A review.” *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[46] M. Arifuzzaman and E. Arslan, “Online optimization of file transfers in high-speed networks,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[47] E. Arslan, K. Guner, and T. Kosar, “Harp: Predictive transfer optimization based on historical analysis and real-time probing,” in *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 288–299.

[48] E. Arslan and T. Kosar, “High-speed transfer optimization based on historical analysis and real-time tuning,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1303–1316, 2018.