End-to-End Network Delay Guarantees for Real-Time Systems using SDN

Rakesh Kumar[†], Monowar Hasan[†], Smruti Padhy[†*], Konstantin Evchenko[†],
Lavanya Piramanayagam[||], Sibin Mohan[†] and Rakesh B. Bobba[§]
[†]University of Illinois at Urbana-Champaign, USA, [||]PES University, India, [§]Oregon State University, USA
Email: [†]{kumar19, mhasan11, evchenk2, sibin}@illinois.edu, [*]smruti@mit.edu,
[||]lava281995@gmail.com, [§]rakesh.bobba@oregonstate.edu

***Abstract*—Real-time systems (RTS) require end-to-end delay guarantees for the delivery of network packets. In this paper, we propose a framework to reduce the management and integration overheads for such real-time (RT) network flows by leveraging the capabilities of software-defined networking (SDN) – capabilities that include global visibility and management of the network. Given the specifications of flows that must meet hard real-time requirements, our framework synthesizes paths through the network. To guarantee that these flows meet both, their bandwidth and end-to-end timing requirements, our framework solves a multi-constraint optimization problem using a heuristic algorithm. We use exhaustive emulations and experiments on hardware switches to demonstrate our techniques and feasibility of our approach. As a result of this work, SDNs become "delay-aware" and thus can be adapted for use in safety-critical and other delay-sensitive applications.**

I. Introduction

Software-defined networking (SDN) [1] has become increasingly popular since it allows for better management of network resources, application of security policies and testing of new algorithms and mechanisms. It finds use in a wide variety of domains – from enterprise systems [2] to cloud computing services [3], from military networks [4] to power systems [5] [6], among others. The global view of the network obtained by the use of SDN architectures provides significant advantages when compared to traditional networks. It allows designers to push down rules to the various nodes in the network that can, to a fine level of precision, manage the bandwidth and resource allocation for flows through the entire network. However, current SDN architectures do *not* reason about end-to-end delay experienced by individual flows. On the other hand, real-time systems (RTS), especially those with stringent timing constraints, need to reason about such delay. *Packets must be delivered between hosts with guaranteed upper bounds on delay*. Examples of such systems include avionics, automobiles, industrial control systems, power substations, manufacturing plants, *etc.*

The material in this paper is based upon work supported in part by the U.S. Department of Energy (DoE) awards DE-OE0000679 and DE-OE0000780 and National Science Foundation (NSF) grant number NSF-CPS-1544901. Any findings, opinions, recommendations or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

[*]Smruti Padhy was affiliated with University of Illinois at the time of this work but her affiliation has since changed to Massachussetts Institute of Technology.

While RTS can include different types of traffic[1], in this paper we focus on the high priority flows that have stringent timing requirements, predefined priority levels and can tolerate little to no loss of packets. We refer to such traffic as "Class I" traffic. Typically, in many safety-critical RTS, the properties of all Class I flows are well known, *i.e.,* designers will make these values available ahead of time. Any changes (addition/removal of flows or modifications to the timing or bandwidth requirements) often requires a serious system redesign. The number (and properties) of other flows could be more dynamic – consider the on-demand video situation in an airplane where new flows could arise and old ones stop based on the viewing patterns of passengers.

Current safety-critical systems often have separate networks (hardware and software) for different classes of flows (for safety and sometimes security reasons). This leads to significant overheads (equipment, management, weight, *etc.*) and also potential for errors/faults and even increased attack surfaces and vectors. Existing systems, *e.g.,* avionics full-duplex switched Ethernet (AFDX) [7]–[9], controller area network (CAN) [10], *etc.* that are in use in many of these domains are either proprietary, complex, expensive or might even require custom hardware. Despite the fact that AFDX switches ensure timing determinism, packets transmitted on such switches may be changed frequently at run-time when sharing resources (*e.g.,* bandwidth) among different networks [11]. In such situations, a dynamic configuration is required to route packets based on switch workloads and flow delays to meet all the high priority Quality of Service (QoS) requirements (*e.g.,* end-to-end delay). In addition AFDX protocols require custom hardware [12].

In this paper we present mechanisms to *guarantee end-to-end delays for high-criticality flows (Class I) on networks constructed using SDN switches.* The advantage of using SDN is that it provides a centralized mechanism for developing and managing the system. The global view is useful in providing the end-to-end guarantees that are required. Another advantage

[1]For instance, *(a)* high priority/criticality traffic that is essential for the correct and safe operation of the system; *(b)* medium criticality traffic that is critical to the correct operation of the system, but with some tolerances in delays, packet drops, *etc.*; and *(c)* low priority traffic – essentially all other traffic in the system that does not really need guarantees on delays or bandwidth such as engineering traffic in power substations, multimedia flows in aircraft, *etc.*

is that the hardware/software resources needed to implement all of the above types of traffic can be reduced since we can use the same network infrastructure (instead of separate ones as is the case currently). On the other hand, the current standards used in traditional SDN (OpenFlow [1], [13]) generally do not support end-to-end delay guarantees or even existing real-time networking protocols such as AFDX. Retrofitting OpenFlow into AFDX is not straightforward and is generally less effective [14].

A number of issues arise while developing a software-defined networking infrastructure for use in real-time systems. For example, the Class I flows need to meet their *timing* (*e.g.,* end-to-end delay) requirements for the real-time system to function correctly. In order for these flows to meet their requirements in the presence of other non-critical traffic, we need to *find a path* through the network, along with necessary resources. However, current SDN implementations reason about resources like bandwidth instead of delays. Hence, we devise a mechanism to extend the SDN infrastructure to reason about delays for use in RTS. Further, in contrast to traditional SDNs, it is *not necessary to find the shortest path* through the network. Oftentimes, Class I flows can arrive *just in time* [15], [16], *i.e.,* just before their deadline – there is no real advantage in getting them to their destinations well ahead of time. Path layout for real-time SDN is a *non-trivial* problem since, *(i)* we need to understand the delay(s) caused by individual nodes (*e.g.,* switches) on a Class I flow and *(ii)* compose them along the delays/problems caused by the presence of other flows in that node as well as the network in general.

As mentioned earlier, existing SDN systems can reason about bandwidth and/or the number of hops in a network. In our work, we reason about strict end-to-end delays for admission control since simply apportioning the network based on bandwidth/number of hops *cannot* guarantee that the flows will meet their timing requirements (especially in the face of other traffic). Also, we try to maximize the number of such flows (that require end-to-end delays) that can fit into the given network – using a heuristic method to solve a multi-constraint optimization problem. We then realize the routes accepted by the admission policy by over-provisioning (in terms of one queue per real-time flow) – this circumvents the issue of queuing delays (and hence avoids the lack of determinism that is required for hard real-time systems).

In this work[2] we consider Class I (*i.e.,* high-criticality) flows and develop a scheme to meet their timing constraints[3]. The main contributions of this work are summarized as follows:

1) We motivate the need for isolating flows into different queues to provide stable end-to-end delays (Section III-A) even in the presence of other types of traffic in the system.
2) We developed mechanisms to guarantee delay constraints for individual end-to-end flows in hard real-time systems based on COTS SDN hardware (Sections III, IV and V) by formulating a multi-constraint problem.

We empirically evaluate the effectiveness of the proposed approach with various topologies and UDP traffic (Section VII) on a widely-used emulation platform. Our results demonstrate that the end-to-end delay experienced by the critical class-I flows falls within user their specified timing deadline.

II. Background

1) The Software Defined Networking Model: In traditional networking architectures, control and data planes coexist on network devices. SDN architectures simplify access the system by logically centralizing the control-plane state into *controller* (see Figure 1). This programmable and centralized state then drives the network devices that perform homogeneous forwarding plane functions [18] and can be modified to control the behavior of the SDN in a flexible manner.

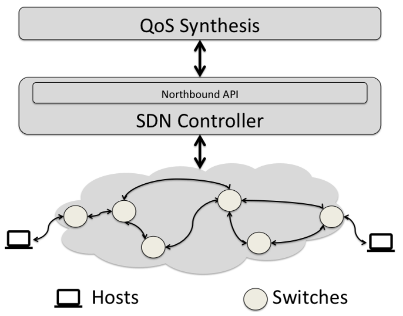


Fig. 1. An SDN with a six switch topology. Each switch also connects to the controller via a management port (not shown). The QoS Synthesis module (Section VI) synthesizes flow rules by using the northbound API.

In order to construct a logically centralized *state* of the SDN system, the controller uses management ports to gauge the current topology and gather data-plane state from each switch. This state is then made available through a *northbound* API to be used by the applications. An application (*e.g.,* our prototype proposed in this paper) uses this API to obtain a snapshot of the SDN state. This state also includes the network topology.

2) The Switch: An SDN switch consists of a table processing pipeline and a collection of physical *ports*. Packets arrive at one of the ports and they are processed by the pipeline made up of one or more flow tables. Each flow table contains *flow rules* ordered by their priority. Each flow rule represents an atomic unit of decision-making in the control-plane. During the processing of a single packet, *actions* (*e.g.,* decision-making entities) can modify the packet, forward it out of the switch or drop it.

When a packet arrives at a switch, it is compared with flow rules in one or more flow table pipelines. In a given table, the contents of the packet header are compared with the flow rules in decreasing order of rule priority. When a matching flow rule is found, the packet is assigned a set of actions specified by the flow rule to be applied at the end of table processing pipeline. Each flow rule comprises of two parts:

- **Match**: is set of packet header field values that a given flow rule applies to. Some are characterized by single

[2]A preliminarily version of the work was presented to the 2017 RTN workshop [17]. In this paper we extend the workshop version with more comprehensive experiments (Section VII) and evaluation on hardware switches (Section III-A).

[3]We will work on integrating other types of traffic in future work.

values (*e.g.,* VLAN ID: 1, or UDP Destination Port: 80), others by a range (*e.g.,* Destination IP Addresses: 10.0.0.0/8). If a packet header field is not specified then it is considered to be a wild card.

- **Instructions Set**: is a set of actions applied by the flow rule to a matching packet. The actions can specify the egress port (OutputPort) for packets matching the rule. Furthermore, in order to make the appropriate allocation of bandwidth for the matching packets, the OpenFlow [13] specification provides two mechanisms:
  - **Queue References**: Every OpenFlow switch is capable of providing isolation to traffic from other flows by enqueuing them on separate queues on the egress port. Each queue has an associated QoS configuration that includes, most importantly, the service rate for traffic that is enqueued in it. The OpenFlow standard itself does not provide mechanisms to configure queues; however, each flow rule can refer to a specific queue number for a port, besides the OutputPort.
  - **Meters**: Beyond the isolation provided by using queues, OpenFlow switches are also capable of limiting the rate of traffic in a given network flow by using objects called meters. Meters on a switch are stored in a meter table and can be added/deleted by using messages specified in OpenFlow specification. Each meter has an associated metering rate. Each flow rule can refer to only a single meter.

III. System Model

Consider an SDN topology (N) with open flow switches and controller and a set of real-time flows (F) with specified delay and bandwidth guarantee requirements. The *problem is to find paths for the flows (through the topology) such that the flow requirements (i.e., end-to-end delays) can be guaranteed for the maximum number of critical flows*. We model the network as an undirected graph $N(V, E)$ where V is the set of nodes, each representing a switch port in a given network and E is set of the edges[4], each representing a possible path for packets to go from one switch port to another. Each port $v \in V$ has a set of queues v_q associated with it, where each queue is assigned a fraction of bandwidth on the edge connected to that port.

Consider a set F of unidirectional, real-time flows that require delay and bandwidth guarantees. The flow $f_k \in F$ is given by a four-tuple (s_k, t_k, D_k, B_k), where $s_k \in V$ and $t_k \in V$ are ports (the source and destination respectively) in the graph, D_k is the maximum delay that the flow can tolerate and B_k is the maximum required bandwidth by the flow. We assume that flow priorities are distinct and the flows are prioritized based on a *"delay-monotonic"* scheme *viz.,* the end-to-end delay budget represents higher priority (*i.e.,* $pri(f_i) > pri(f_j)$ if $D_i < D_j,\ \forall f_i, f_j \in F$ where $pri(f_k)$ represents priority of f_k).

For a flow to go from the source port s_k to a destination port t_k, it needs to traverse a sequence of edges, *i.e.,* a flow path $\mathcal{P}_k$. The problem then, is to synthesize flow rules that use queues at each edge $(u, v) \in \mathcal{P}_k$ that can handle *all* flows F in the given system while still meeting each flow's requirement. If $d_{f_k}(u, v)$ and $b_{f_k}(u, v)$ is the delay faced by the flow and bandwidth assigned to the flow at each edge $(u, v) \in E$ respectively, then $\forall f_k \in F$ and $\forall (u, v) \in \mathcal{P}_k$ the following constraints need to be satisfied:

$$\sum_{(u,v)\in\mathcal{P}_k} d_{f_k}(u, v) \le D_k, \quad \forall f_k \in F \tag{1}$$

$$b_{f_k}(u, v) \ge B_k, \quad \forall (u, v) \in \mathcal{P}_k, \forall f_k \in F. \tag{2}$$

This problem needs to be solved at two levels:

- *Level 1*: Finding the path layout for each flow such that it satisfies the flows' delay and bandwidth constraints. We formulate this problem as a multi-constrained path (MCP) problem and describe the solution in Sections IV and V.
- *Level 2*: Mapping the path layouts from Level 1 on to the network topology by using the mechanisms available in OpenFlow. We describe details of our approach in Section VI.

In addition to the aforementioned delay and bandwidth constraints (see Eqs. (1) and (2)), we need to map flows assigned to a port to the queues at the actual ports. Two possible approaches are: *(a) allocate each flow to an individual queue* or *(b) multiplex flows onto a smaller set of queues* and dispatch the packets based on priority. In fact, as we illustrate in the following section, the queuing approach used will impact the delays faced by the flows at each link. Our intuition is that the *end-to-end delays are lower and more stable* when *separate queues* are provided to each critical flow – especially as the rates for the flows get closer to their maximum assigned rates. Given the deterministic nature of many RTS, the number of critical flows are often limited and well defined (*e.g.,* known at design time). Hence, such over-provisioning is an acceptable design choice – from computing power to network resources (for instance one queue per critical real time flow). We carried out some experiments to demonstrate this (and to highlight the differences between these two strategies) – this is outlined in the following section.

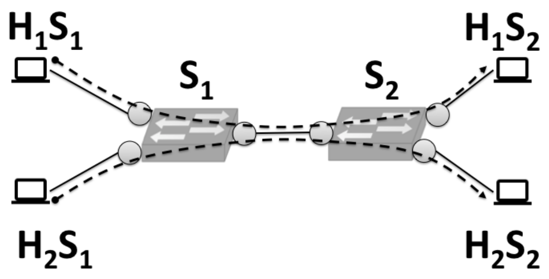
A. Queue Assignment Strategies

We intend to synthesize configurations for Class I traffic such that it ensures *complete isolation of packets for each designated class I flow at each switch in its path.*

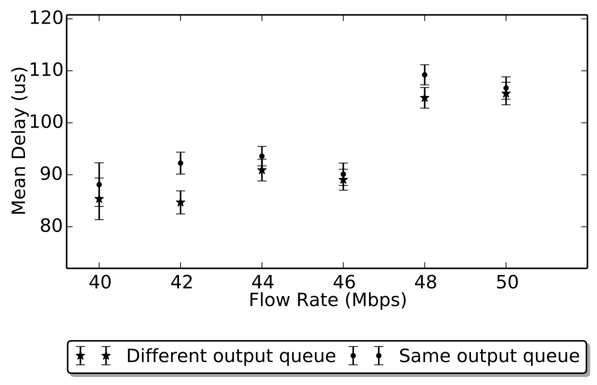
In order to test how using output queues can provide isolation to flows in a network so that each can meet its delay and bandwidth requirements simultaneously, we performed experiments using OpenFlow enabled hardware switches[5]. The experiments use a simple topology that contains two white box Pica8 P-3297 [20] switches (`s1`, `s2`) connected via a single link as shown in Figure 2(a). Each switch has two hosts connected to it. Each host is a Raspberry Pi 3 Model B [21] running Raspbian Linux.

[4]We use the terms *edge* and *link* interchangeably throughput the paper.

[5]We also conduct similar experiments with software emulations (*e.g.,* by using Mininet [19] topology) and observe similar trends (see Appendix B).



(a)



(b)

Fig. 2. Delay measurement experiments: (a) The two-switch, four host topology used in the experiments with the active flows. (b) The measured mean and 99^{th} percentile per-packet delay for the packets in the active flows in 30 iterations.

We configured flow rules and queues in the switches to enable connectivity among hosts at one switch with the hosts at other switch. We experimented with two ways to queue the packets as they cross the switch-to-switch link: *(i)* in one case, we queue packets belonging to the two flows *separately* in two queues (*i.e.,* each flow gets its own queue), each configured at a maximum rate of 50 Mbps *(ii)* in the second case, we queue packets from both flows in the *same queue* configured at a maximum rate of 100 Mbps.

After configuring the flow rules and queues, we used `netperf` [22] to generate the following packet flows: the first starting at the host `h1s1` destined to host `h1s2` and the second starting at host `h2s1` with a destination host `h2s2`. Both flows are identical and are triggered simultaneously to last for 15 seconds. We changed the rate at which the traffic is sent across both flows to measure the average per-packet delay. Figure 2(b) plots the average value and standard error over 30 iterations. The x-axis indicates the rate at which the traffic is sent via `netperf`, while the y-axis shows the average per-packet delay. The following key observations stand out:

1) The per-packet average delay increases in both cases as traffic send rate approaches the configured rate of 50 Mbps. This is an expected queue-theoretic outcome and motivates the need for slack allocations for all applications in general. For example, if an application requires a bandwidth guarantee of 1 Mbps, it should be allocated 1.1 Mbps for minimizing jitter.
2) The case with separate queues experiences lower average per-packet delay when flow rates approach the maximum rates. We observed this effect even more strongly in the case of software switches (Appendix B). This indicates that when more than one flow uses the same queue, there is interference caused by both flows to each other. This becomes a source of unpredictability and eventually may cause the end-to-end delay guarantees for the flows to be not met or perturbed significantly.

Thus, isolating flows using separate queues results in lower and more stable delays especially when traffic rate in the flow approaches the configured maximum rates. Such isolation leads to a correct-by-design approach that ensures that each flow is allocated bandwidth at each switch such that it does not experience queueing delays. The maximum processing delay along a single link can be measured and used as input to a path allocation algorithm that we describe in the following section.

IV. Path Layout: Overview and Solution

We now present a more detailed version of the problem (composing paths that meet end-to-end delay constraints for critical real-time flows) and also an overview of our solution.

Problem Overview: Let $\mathcal{P}_k$ be the path from s_k to t_k for flow f_k that needs to be determined. Let $\mathfrak{D}(u,v)$ be the delay incurred on the edge $(u,v) \in E$. The total delay for f_k over the path $\mathcal{P}_k$ is given by

$$\mathfrak{D}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \mathfrak{D}(u,v). \tag{3}$$

Therefore we define the following constraint on end-to-end delay for the flow f_k as

$$\mathfrak{D}_k(\mathcal{P}_k) \leq D_k. \tag{4}$$

Note that the end-to-end delay for a flow over a path has following delay components[6]: *(a)* processing time of a packet at a switch, *(b)* propagation on the physical link, *(c)* transmission of packet over a physical link, and *(d)* queuing at the ingress/egress port of a switch. As discussed in the Section III, we use separate queues for each flow with assigned required rates. We also slightly overprovision the bandwidth for such flows so that critical real-time flows do not experience queueing delays. Hence, we consider queuing delays to be negligible. We discuss how to obtain the values of other components of delay in Appendix A.

The second constraint that we consider in this work is *bandwidth utilization*, that for an edge (u,v) for a flow f_k, can be defined as:

$$\mathfrak{B}_k(u,v) = \frac{B_k}{B_e(u,v)} \tag{5}$$

[6]While network links are non-preemptive, we model the time for transmitting the largest packet as part of the amortized processing delay that can be calculated for a given switch (see Appendix A). As such, our method can incorporate any method to estimate such delays and is analogous to scheduling algorithms that can benefit from, yet are completely separate from the mechanisms for, worst-case execution time (WCET) analysis.

where B_k is the bandwidth requirement of f_k and $B_e(u,v)$ is *residual* (*viz.,* available) bandwidth of an edge $(u,v) \in E$. Therefore, bandwidth utilization over a path ($\mathcal{P}_k$), for a flow f_k is defined as:

$$\mathfrak{B}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \mathfrak{B}_k(u,v). \tag{6}$$

Note that the bandwidth utilization over a path $\mathcal{P}_k$ for flow f_k is bounded by

$$\mathfrak{B}_k(\mathcal{P}_k) \le \max_{(u,v)\in E} \mathfrak{B}_k(u,v)|V|. \tag{7}$$

where $|V|$ is the cardinality of a set of nodes (ports) in the topology N. Therefore in order to ensure that the bandwidth requirement B_k of the flow f_k is guaranteed, it suffices to consider the following constraint on bandwidth utilization

$$\mathfrak{B}_k(\mathcal{P}_k) \le \widehat{B}_k \tag{8}$$

where $\widehat{B}_k = \max_{(u,v)\in E} \mathfrak{B}_k(u,v)|V|$.

Remark 1. *The selection of an optimal path for each flow $f_k \in F$ subject to delay and bandwidth constraints in Eq. (4) and (8), respectively can be formalized as a multi-constrained path (MCP) problem that is known to NP-complete [23].*

Therefore we extend a polynomial-time heuristic similar to that presented in literature [24] and apply for our context (Algorithm 2). The key idea is to *relax* one constraint (*e.g.,* delay or bandwidth) at a time and try to obtain a solution. If the original MCP problem has a solution, one of the relaxed versions of the problem will also have a solution [24]. In what follows, we briefly describe the polynomial-time solution for the path layout problem.

Polynomial-time Solution to the Path Layout Problem: Let us represent the delay and bandwidth constraint as follows

$$\widetilde{\mathfrak{D}}_k(u,v) = \left\lceil \frac{X_k \cdot \mathfrak{D}(u,v)}{D_k} \right\rceil \tag{9}$$

$$\widetilde{\mathfrak{B}}_k(u,v) = \left\lceil \frac{X_k \cdot \mathfrak{B}_k(u,v)}{\widehat{B}_k} \right\rceil \tag{10}$$

where X_k is a given positive integer. For instance, if we relax the bandwidth constraint (*e.g.,* represent $\mathfrak{B}_k(\mathcal{P}_k)$ in terms of $\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{B}}_k(u,v)$), Eq. (8) can be rewritten as

$$\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) \le X_k. \tag{11}$$

Besides, the solution to this relaxed problem will also be a solution to the original MCP [24]. Likewise, if we relax the delay constraint, Eq. (4) can be rewritten as

$$\widetilde{\mathfrak{D}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{D}}_k(u,v) \le X_k. \tag{12}$$

Let the variable $d_k[v,i]$ preserve an *estimate* of the path from s_k to t_k for $\forall v \in V$, $i \in \mathbb{Z}^+$ (refer to Algorithm 1). There exists a solution (*e.g.,* a path $\mathcal{P}_k$ from s_k to t_k) if *any* of the two conditions is satisfied when the *original MCP problem is solved by the heuristic*.

- *When the bandwidth constraint is relaxed:* The delay and (relaxed) bandwidth constraints, *e.g.,* $\mathfrak{D}_k(\mathcal{P}_k) \le D_k$ and $\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) \le X_k$ are satisfied if and only if

$$d_k[t,i] \le D_k, \quad \exists i \in [0, X_k] \wedge i \in \mathbb{Z}.$$

- *When the delay constraint is relaxed:* The (relaxed) delay and bandwidth constraints, *e.g.,* $\widetilde{\mathfrak{D}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{D}}_k(u,v) \le X_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \le \widehat{B}_k$ are satisfied if and only if

$$d_k[t,i] \le X_k, \quad \exists i \in [0, \widehat{B}_k] \wedge i \in \mathbb{Z}.$$

V. Algorithm Development

A. Path Layout

Our proposed approach is based on a polynomial-time solution to the MCP problem presented in literature [24]. Let us consider `MCP_HEURISTIC`$(N, s, t, W_1, W_2, C_1, C_2)$, an instance of polynomial-time heuristic solution to the MCP problem that finds a path $\mathcal{P}$ from s to t in any network N, satisfying constraints $W_1(\mathcal{P}) \le C_1$ and $W_2(\mathcal{P}) \le C_2$.

The proposed heuristic solution of MCP problem, as summarized in Algorithm 1 works as follows. Let

$$\Delta(v,i) = \min_{\mathcal{P}\in P(v,i)} W_1(\mathcal{P}) \tag{13}$$

where $P(v,i) = \{\mathcal{P} \mid W_2(\mathcal{P}) = i, \mathcal{P} \text{ is any path from } s \text{ to } t\}$ is the smallest $W_1(\mathcal{P})$ of those paths from s to v for which $W_2(\mathcal{P}) = C_2$. For each node $v \in V$ and each integer $i \in [0, \cdots, C_2]$ we maintain a variable $d[v,i]$ that keeps an estimation of the smallest $W_1(\mathcal{P})$. The variable initialized to $+\infty$ (Line 3), which is always greater than or equal to $\delta(v,i)$. As the algorithm executes, it makes better estimation and eventually reaches $\Delta(v,i)$ (Line 8-15). Line 3-17 in Algorithm 1 is similar to the single-cost path selection approach presented in earlier work [24, Sec. 2.2] and for the purposes of this work, we have extended the previous approach for our formulation.

We store the path in the variable $\pi[v,i], \forall v \in V, \forall i \in [0, \cdots, C_2]$. When the algorithm finishes the search for path (Line 17), there will be a solution if and only if the following condition is satisfied [24]

$$\exists i \in [0, \cdots, C_2], \quad d[t,i] \le C_1. \tag{14}$$

If it is not possible to find any path (*e.g.,* the condition in Eq. (14) is not satisfied), the algorithm returns False (Line 41). If there exists a solution (Line 19), we extract the path by backtracking (Line 21-29). Notice that the variable $\pi[v,i]$ keeps the immediate preceding node of v on the path (Line 13). Therefore, the path can be recovered by tracking π starting from destination t through all immediate nodes until reaching the source s. Based on this MCP abstraction, we developed a path selection scheme considering delay and bandwidth constraints (Algorithm 2) that works as follows.

For each flow $f_k \in F$, starting with highest (*e.g.,* the flow with tighter delay requirement) to lowest priority, we first keep the delay constraint unmodified and relax the bandwidth constraint by using Eq. (10) and solve `MCP_HEURISTIC`$(N, s_k, t_k, \mathfrak{D}_k, \widetilde{\mathfrak{B}}_k, D_k, X_k)$ (Line 3) using

Algorithm 1 Multi-constraint Path Selection

Input: The network $N(V,E)$, source s, destination t, constraints on links $W_1 = [w_1(u,v)]_{\forall(u,v)\in E}$ and $W_2 = [w_2(u,v)]_{\forall(u,v)\in E}$, and the bounds on the constraints $C_1 \in \mathbb{R}^+$ and $C_2 \in \mathbb{R}^+$ for the path from s to t.

Output: The path $\mathcal{P}^*$ if there exists a solution (*e.g.,* $W_1(\mathcal{P}^*) \leq C_1$ and $W_2(\mathcal{P}^*) \leq C_2$), or False otherwise.

```
 1: function MCP_HEURISTIC($N, s, t, W_1, W_2, C_1, C_2$)
 2:     /* Initialize local variables */
 3:     $d[v,i] := \infty, \pi[v,i] := NULL, \quad \forall v \in V, \ \forall i \in [0,C_2] \wedge i \in \mathbb{Z}$
 4:     $d[s,i] := 0 \ \ \forall i \in [0,C_2] \wedge i \in \mathbb{Z}$
 5:     /* Estimate path */
 6:     for $i \in |V|-1$ do
 7:         for each $j \in [0,C_2] \wedge j \in \mathbb{Z}$ do
 8:             for each edge $(u,v) \in E$ do
 9:                 $j' := j + w_2(u,v)$
10:                 if $j' \leq C_2$ and $d[v,j'] > d[u,j] + w_1(u,v)$ then
11:                     /* Update estimation */
12:                     $d[v,j'] := d[u,j] + w_1(u,v)$
13:                     $\pi[v,j'] := u$ /* Store the possible path */
14:                 end if
15:             end for
16:         end for
17:     end for
18:     /* Check for solution */
19:     if $d[t,i] \leq C_1$ for $\exists i \in [0,C_2] \wedge i \in \mathbb{Z}$ then
20:         /* Solution found, obtain the path by backtracking */
21:         $\mathcal{P} := \emptyset$, $done :=$ False, $currentNode := t$
22:         /* Find the path from $t$ to $s$ */
23:         while not $done$ do
24:             for each $j \in [0,C_2] \wedge j \in \mathbb{Z}$ do
25:                 if $\pi[currentNode, j]$ not $NULL$ then
26:                     add $currentNode$ to $\mathcal{P}$
27:                     if $currentNode = s$ then
28:                         $done :=$ True /* Backtracking complete */
29:                         break
30:                     end if
31:                     /* Search for preceding hop */
32:                     $currentNode := \pi[currentNode, j]$
33:                     break
34:                 end if
35:             end for
36:         end while
37:         /* Reverse the list to obtain a path from $s$ to $t$ */
38:         $\mathcal{P}^* :=$ reverse($\mathcal{P}$)
39:         return $\mathcal{P}^*$
40:     else
41:         return False /* No Path found that satisfies $C_1$ and $C_2$ */
42:     end if
43: end function
```

Algorithm 2 Layout Path Considering Delay and Bandwidth Constraints

Input: The network $N(V,E)$, set of flows F, delay and bandwidth utilization constraints on links $\mathfrak{D}_k = [\mathfrak{D}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{D}}_k = [\widetilde{\mathfrak{D}}_k(u,v)]_{\forall(u,v)\in E}$ and $\mathfrak{B}_k = [\mathfrak{B}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{B}}_k = [\widetilde{\mathfrak{B}}_k(u,v)]_{\forall(u,v)\in E}$, for each flow $f_k \in F$, respectively, and the delay and bandwidth bounds $D_k \in \mathbb{R}^+$ and $\widehat{B}_k \in \mathbb{R}^+$, respectively, and positive constant $X_k \in \mathbb{Z}, \forall f_k \in F$.

Output: The path vector $\boldsymbol{\mathcal{P}} = [\mathcal{P}_k]_{\forall f_k \in F}$ where $\mathcal{P}_k$ is the path if the delay and bandwidth constraints (*e.g.,* $\mathfrak{D}_k(\mathcal{P}_k) \leq D_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \leq \widehat{B}_k$) are satisfied for f_k, or False otherwise.

```
 1: for each $f_k \in F$ (starting from higher to lower priority) do
 2:     Discard the the links for which $B_{e'}(u,v) < B_k, \forall e' \in E$
 3:     /* Relax bandwidth constraint and solve */
 4:     Solve MCP_HEURISTIC($N, s_k, t_k, \mathfrak{D}_k, \widetilde{\mathfrak{B}}_k, D_k, X_k$) by using Algorithm 1
 5:     if SolutionFound then /* Path found for $f_k$ */
 6:         /* Add path to the path vector $\boldsymbol{\mathcal{P}}$ */
 7:         $\mathcal{P}_k := \mathcal{P}^*$ where $\mathcal{P}^*$ is the solution obtained by Algorithm 1
 8:     else
 9:         /* Relax delay constraint and try to obtain the path */
10:         Solve MCP_HEURISTIC($N, s_k, t_k, \widetilde{\mathfrak{D}}_k, \mathfrak{B}_k, X_k, \widehat{B}_k$) by using Algorithm 1
11:         if SolutionFound then
12:             /* Path found by relaxing delay constraint */
13:             $\mathcal{P}_k := \mathcal{P}^*$ /* Add path to the path vector */
14:             /* Update remaining available bandwidth */
15:             $B_e(u,v) := B_e(u,v) - B_k, \ \forall(u,v) \in \mathcal{P}_k$
16:         else
17:             $\mathcal{P}_k :=$ False /* Unable to find any path for $f_k$ */
18:         end if
19:     end if
20: end for
```

Algorithm 1. We only consider the feasible links in the topology, *e.g.,* the links with residual bandwidth $B_{e'}(u,v) \geq B_k, \forall e' \in E$.

If a solution exists, the corresponding path $\mathcal{P}_k$ is assigned for f_k (Line 6). However, if relaxing the bandwidth constraint does not return a path, we further relax the delay constraint by using Eq. (9), keeping the bandwidth constraint unmodified and solve MCP_HEURISTIC($N, s_k, t_k, \widetilde{\mathfrak{D}}_k, \mathfrak{B}_k, X_k, \widehat{B}_k$) (Line 9). Once the path is found, we allocate the bandwidth for the scheduled flow and update the residual link bandwidth (Line 15). If the path is not found after *both* relaxation steps, the algorithm returns False (Line 17) since it is not possible to assign a path for f_k such that both delay and bandwidth constraints are satisfied. Note that the heuristic solution of the MCP depends of the parameter X_k. From our experiments we find that if a solution exists, the algorithm is able to find a path as long as $X_k \geq 10$.

B. Complexity Analysis

Note that Line 8 in Algorithm 1 is executed at most $(C_2+1)(V-1)E$ times. Besides, if there exists a path, the worst-case complexity to extract the path is $|\mathcal{P}|C_2$. Therefore, time complexity of Algorithm 1 is $O(C_2(VE + |\mathcal{P}|)) = O(C_2VE)$. Hence the worst-case complexity (*e.g.,* when both of the constraints need to be relaxed) to execute Algorithm 2 for each flow $f_k \in F$ is $O((X_k + \widehat{B}_k)VE)$.

VI. Implementation

We implement our prototype as an *application that uses the northbound API* for the Ryu controller [25]. The prototype application accepts the specification of flows in the SDN. The flow specification contains the classification, bandwidth requirement and delay budget of each individual flow. In order for a given flow f_k to be realized in the network, the control-plane state of the SDN needs to be modified. The control-plane needs to route traffic along the path calculated for each f_k as described in Section V. In this section, we describe how this is accomplished by decomposing the network-wide state modifications into a set of smaller control actions (called Intents) that occur at each switch.

A. Forwarding Intent Abstraction

An *intent* represents the *actions performed on a given packet at each individual switch*. Each flow f_k is decomposed into a set of intents as shown in Figure 3. The number of

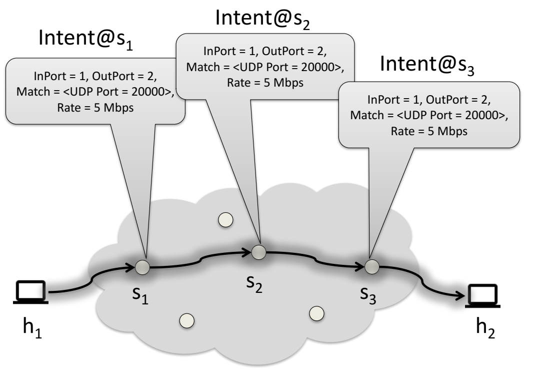

Intent@s_1
InPort = 1, OutPort = 2, Match = <UDP Port = 20000>, Rate = 5 Mbps
Intent@s_2
InPort = 1, OutPort = 2, Match = <UDP Port = 20000>, Rate = 5 Mbps
Intent@s_3
InPort = 1, OutPort = 2, Match = <UDP Port = 20000>, Rate = 5 Mbps
s_1
s_2
s_3
h_1
h_2

Fig. 3. Illustration of decomposition of a flow f_k into a set of intents: f_k here is a flow from the source host h_1 to the host h_2 carrying mission-critical DNP3 packets with destination UDP port set to $20,000$. In this example, each switch that f_k traverses has exactly two ports.

intents that are required to express actions that the network needs to perform (for packets in a flow) is the same as the number of switches on the flow path. Each intent is a tuple given by (Match, InputPort, OutputPort, Rate). Here, Match defines the set of packets that the intent applies to, InputPort and OutputPort are where the packet arrives and leaves the switch and finally, the Rate is intended data rate for the packets matching the intent. In our implemented mechanism for laying down flow paths, each intent translates into a single OpenFlow [13] flow rule that is installed on the corresponding switch in the flow path.

B. Bandwidth Allocation for Intents

In order to guarantee bandwidth allocation for a given flow f_k, each one of its intents (at each switch) in the path must allocate the same amount of bandwidth. As described above, each intent maps to a flow rule and the flow rule can refer to a meter, queue or both. However, meters and queues are limited resources. Also not all switch implementations provide both of them. As mentioned earlier (Section III), we use the strategy of one queue per flow that guarantees better isolation among flows and results in stable delays.

C. Intent Realization

Each intent is realized by installing a corresponding flow rule by using the northbound API of the Ryu controller. Other than using the intent's Match and OutputPort, these flow rules refer to corresponding queues and/or meters. If meters are used, then they are also synthesized by using the controller API. However, OpenFlow does not support installation of queues in its controller-switch communication protocol, hence the queues are installed separately by interfacing directly with the switches by using a switch API or command line interface.

VII. Evaluation

The goal of the evaluation in this section is two-fold: *(a)* schedulability of a given set of flows across various topologies to explore the design space/performance of the path layout algorithm in Section VII-A, and *(b)* an empirical evaluation, using Mininet, that demonstrates the effectiveness of our end-to-end delay guaranteeing mechanisms even in the presence of other traffic in the network (Section VII-B). The parameters used in the experiments are summarized in Table I.

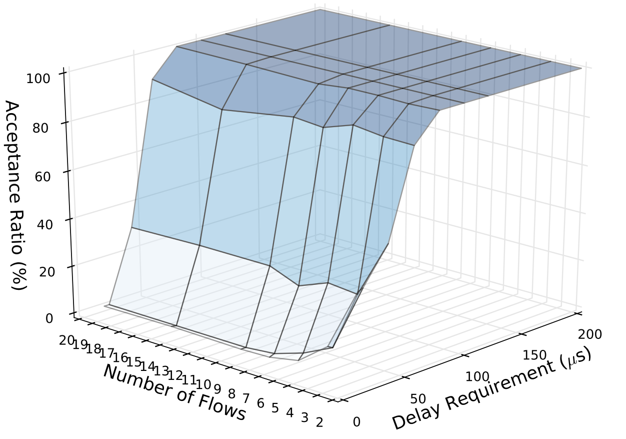


Fig. 4. Schedulability of the flows in different network topology. For each of the (delay-requirement, number-of-flows) pair (*e.g.,* x-axis and y-axis of the figure), we randomly generate 250 different topology. In other words, total 8 × 7 × 250 = 14,000 different topology were tested in the experiments.

TABLE I
EXPERIMENTAL PLATFORM AND PARAMETERS

Artifact/Parameter	**Values**
Number of switches	5
Bandwidth of links	10 Mbps
Bandwidth requirement of a flow	[1, 5] Mbps
SDN controller	Ryu 4.7
Switch configuration	Open vSwitch 2.3.0
Network topology	Synthetic/Mininet 2.2.1
OS	Debian, kernel 3.13.0-100

A. Performance of the Path Layout Algorithms

Topology Setup and Parameters: In the first set of experiments we explore the design space (*e.g.,* feasible delay requirements) with randomly generated network topologies and synthetic flows. For each of the experiments we randomly generate a graph with 5 switches and create $f_k \in [2, 20]$ flows. Each switch has 2 hosts connected to it. We assume that the bandwidth of each of the links $(u, v) \in E$ is 10 Mbps (*e.g.,* IEEE 802.3t standard [26]). For this experiment the link delays are randomly generated within [5, 25] μs. For each randomly-generated topology, we consider the bandwidth requirement as $B_k \in [1, 5]$ Mbps, $\forall f_k$.

Results: We say that a given network topology with set of flows is *schedulable* if all the real-time flows in the network can meet the delay and bandwidth requirements. We use the *acceptance ratio* metric (z-axis in Fig. 4) to evaluate the schedulability of the flows. *The acceptance ratio is defined as the number of accepted topologies (e.g., the flows that satisfied bandwidth and delay constraints) over the total number of generated ones.* To observe the impact of delay budgets in different network topologies, we consider the end-to-end delay requirement D_k, $\forall f_k \in F$ as a function of the topology. In particular, for each randomly generated network topology G_i we set the minimum delay requirement for the highest priority flow as $D_{min} = \beta\delta_i$ μs, and increment it by $\frac{D_{min}}{10}$ for each of the remaining flows. Here δ_i is the diameter (*e.g.,* maximum eccentricity of any vertex) of the graph G_i in the i-th spatial realization of the network topology, $\beta = \frac{D_{min}}{\delta_i}$ and D_{min} represents x-axis values of Fig. 4[7]. For each (delay-requirement, number-of-flows) pair, we randomly generate 250 different topologies and measure the acceptance ratios. As Fig. 4 shows, stricter delay requirements (*e.g.,* less than 60 μs for a set of 20 flows) limit the schedulability (*e.g.,* only 60% of the topology is schedulable). Increasing the number of flows limits the available resources (*e.g.,* bandwidth) and thus the algorithm is unable to find a path that satisfies the delay requirements of *all* the flows.

B. Emulation experiments using Mininet

While the flow paths are laid out in a *correct-by-construction* manner (see Algorithm 2), our evaluation in this section tests our algorithms with a variety of cases to demonstrate that our delay-based admission control algorithms work as they are intended to. This is akin to demonstrating the workings and checking the performance of a proven scheduling algorithm with synthetic task sets.

Experimental Setup: The purpose of the experiment is to evaluate whether our controller rules and queue configurations can provide isolation guarantees so that the real-time flows can meet their delay requirement in a practical setup.

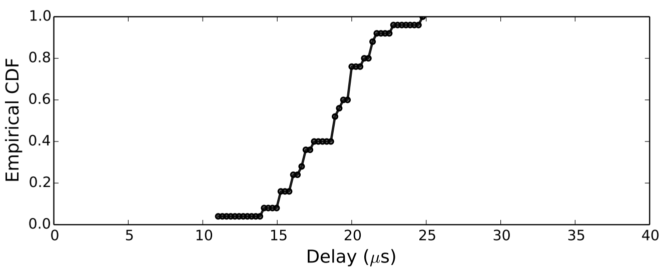
We evaluate the performance of our proposed scheme using Mininet [19] (version 2.2.1) that has widely been used by SDN research community [27]–[31]. Mininet is an open source platform that *emulates* real-world SDN setup by utilizing virtualization on top of a Linux kernel. Mininet has the capability to emulate different kinds of network elements such as host, switches (layer-2), routers (layer-3) and links.

We configured switches using Open vSwitch (OVS) [32] (version 2.3.0) and use Ryu [25] (version 4.7) as our SDN controller. For each of the experiments we randomly generate a Mininet topology using the parameters described in Table I. We develop flow rules in the queues to enable connectivity among hosts in different switches. The packets belonging to the real-time flows are queued separately in individual queues and each of the queues are configured at a maximum rate of $B_k \in [1, 5]$ Mbps. If the host exceeds the configured maximum rate of B_k, our ingress policing throttles the traffic before it enters the switch[8].

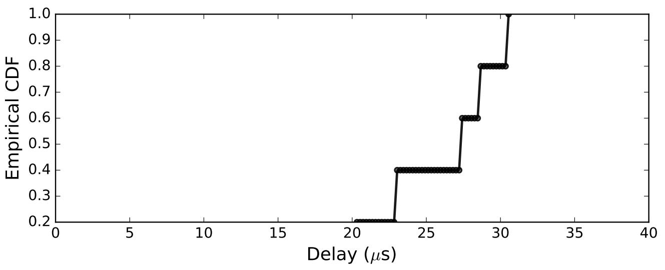
We use `netperf` (version 2.7.0) [22] to generate the UDP traffic [12], [33] between the source and destination for any flow f_k. Once the flow rules and queues are configured, we send packets from source s_k to host t_k for each of the flows f_k. The packets are sent in a burst of 5 with 1 ms inter burst time. All packet flows are triggered simultaneously and last for 10 seconds.

[7]Remember our "delay-monotonic" priority assignment where flows with lower end-to-end delays have higher priority.

[8]In real systems, the bandwidths allocation would be overprovisioned (as mentioned earlier), our evaluation takes a conservative approach.



(a)



(b)

Fig. 5. The empirical CDF of: (a) average round-trip delay when using paths generated by MCP and giving each flow its own queue. worst-case round-trip delay; (b) average round-trip delay experienced when using shortest paths and a single queue. We set the number of flows $f_k = 7$ and examine $7 \times 25 \times 5$ packet flows (each for 10 seconds) to obtain the experimental traces.

To measure the effectiveness of our prototype with mixed (*e.g.,* real-time and non-critical) flows, we enable [1,3] non-critical flows in the network. All of the low-criticality flows use a *separate, single queue* and are served in a FIFO manner – it is the "default" queue in OVS. Since many commercial switches (*e.g.,* Pica8 P-3297, HPE FlexFabric 12900E, *etc.*) supports up to 8 queues per port (and 52 ports per switch), in our Mininet experiments we limit the maximum number of real-time flows to 7. We performed experiments for a single port where each of the 7 real-time flows uses a separate queue and the remaining 8th queue is used for non-critical flows. Our flow rules isolate the non-critical flows from real-time flows. All the experiments are performed on an Intel Xeon 2.40 GHz CPU and Linux kernel version 3.13.0-100.

We assume flows are indexed based on priority, *i.e.,* $D_1 < D_2 < \cdots < D_{|F|}$ and randomly generate 25 different network topologies. We set $D_1 = 10\delta_i$ μs and increment with $\frac{D_1}{10}$ for each of the flow $f_k \in F, k > 1$ where δ_i is the diameter of the graph G_i in the i-th spatial realization of the network topology. For each topology, we randomly generate the traffic with required bandwidth $B_k \in [1, 5]$ Mbps and send packets between source (s_k) and destination (t_k) hosts for 5 times (each transmission lasts for 10 seconds) and log the worst-case round-trip delay experienced by any flow. We define the *expected delay bound* as the expected delay if the packets are routed through the diameter (*i.e.,* the greatest distance between any pair of hosts) of the topology and given by $\mathfrak{D}_i(u, v) \times \delta_i$ where $\mathfrak{D}_i(u, v) = 5$ μs is the delay between the link (u, v) in i-th network realization (refer to Appendix A for the calculation of link delay parameters).

Experience and Evaluation: Recall that we use a correct-by-design principle to lay out the flows in the network. Fig

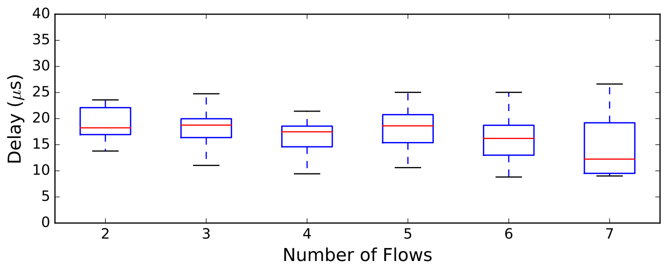


Fig. 6. End-to-end average round-trip delay with varying number of flows. For each set of flow $f_k \in [2, 7]$, we examine $f_k \times 25 \times 5$ packet flows (each for 10 seconds). The blue boxes represent inter-quartile range (*e.g.,* 50% of values for the group) while the inside red lines indicate median value. The upper and lower whiskers represent values outside the middle 50%.

5(a) illustrates the results for the schedulable flows (*viz.,* the set of flows for which *both* delay and bandwidth constraints are satisfied). The y-axis of Fig. 5(a) represents the empirical CDF of average round-trip delay experienced by any flow. From our experiments we find that, the non-critical flows *do not* affect the delay experienced by the real-time flows and the average delay experienced by the real-time flows *always* meets their delay requirements. This is because our flow rules and queue configurations isolate the real-time flows from the non-critical traffic. As seen in Fig. 5(a), the average round-trip delays are less than the maximum expected round-trip delay bound (*e.g.,* $2 \times 5 \times 4 = 40$ μs).

To contrast, we conducted an experiment of laying out the same set of flows but *without* our mechanisms in place. This experiment used shortest-path routing and did not separate the queues for any flows (in contrast to the separate queues for real-time flows in our work). Figure 5(b) plots the empirical CDF of the mean delays experienced by the real-time flows in this setting. As the plot shows, these flows experienced higher and more variable latency than when our mechanisms were in place, thus highlighting the need for the proposed mechanisms being presented in this paper. The 99th-percentile delays were also much higher than when using our mechanisms.

Fig. 6 illustrates the impact of number of flows on the average round-trip delay (represented by y-axis in the figure) with different number of flows (x-axis). Recall that in our experimental setup we assume at most 8 queues per port are available in the switches where 7 real-time flows are assigned to each of 7 queues and the other queue is used for [1, 3] non-critical flows. As shown in Fig. 6, increasing the number of flows slightly decreases quality of experience (in terms of end-to-end delays). With increasing number of packet flows the switches are simultaneously processing forwarding rules received from the controller – hence, it increases the round-trip delay. Recall that the packets of a flow are sent in a bursty manner using `netperf`. Increasing number of flows in the Mininet topology increases the packet loss and thus causes higher delay.

VIII. Discussion

The approach described in this paper leverages the benefits of the SDN architecture to guarantee end-to-end delays in safety-critical hard RTS. Our proposed scheme has some limitations and can be extended in several directions. Most hardware switches limit the maximum number of individual queues[9] that can be allocated to flows. Our current intent realization mechanism reserves one queue per port for each Class I flow and thus can lead to depletion of available queues. However, we found commodity hardware switches (with both meters and queues) that provide 52 ports per switch (with 8 queues per port) and hence may not run out of resources for most RTS application (especially when multiple physical ports are logically combined to form a single virtual port, thus increasing number of queues available along a path). Furthermore, unlike other domains, in RTS, the number of flows are known ahead of time and, based on our proposed method, can be allocated resources accordingly.

Although over-provisioning (in terms of one queue per real-time flow) circumvents the issue of queuing delays, in future work, we intend to overcome this shortcoming by *multiplexing* more than one such flow over a queue, while still meeting the real-time, end-to-end requirements. There are two possible approaches that can be taken. One is to use meters for ingress filtering, but meters themselves are a limited resource. The second is design and implement conditional queueing schemes where a programmable data-plane exposes a mechanism that allows the use of queues based on network conditions instead of it being a static decision. Such mechanisms need to be validated for safety and ultimately they need to ensure that class-I flows shall always meet their timing requirements. Our future work is focused on developing such mechanisms. To do this, we intend to make changes to the "scheduler" inside the switch (that decides what packets go to which queues) and also meters. In addition, changes may be required to the OpenFlow protocol itself so that the controller can track what is happening to the critical (real-time) flows.

Furthermore, in this work we allocate separate queues for each flow and layout paths based on the "delay-monotonic" policy. However establishing and maintaining the flow priority is *not* straightforward if the ingress policing requires sharing queues and ports in the switches. Many existing mechanisms to enforce priority are available in software switches (*e.g.,* the hierarchical token buckets (HTB) in Linux networking stack). In our experience, enabling priority on hardware switches has proven difficult due to firmware bugs.

Finally, we do not impose any admission control policy for the unschedulable (*i.e.,* the flows for which the delay and bandwidth constraints are not satisfied) flows. One approach to enable admission control is to allow m out of k $(m < k)$ packets of a low-priority flow to meet the delay budget by leveraging the concept of (m, k) scheduling [34] in traditional RTS.

IX. Related Work

There have been several efforts to study the provisioning a network such that it meets bandwidth and/or delay constraints

[9] *e.g.,* Pica8 P-3297 and HPE FlexFabric 12900E switches support at most 8 queues.

for the traffic flows. Results from the network calculus (NC) [35] framework offer a concrete way to model the various abstract entities and their properties in a computer network. NC-based models, on the other hand, do not prescribe any formulation of flows that meet given delay and bandwidth guarantees. For synthesis, the NP-complete MCP comes close and Shingang *et al.* formulated a heuristic algorithm [24] for solving MCP. We model our delay and bandwidth constraints based on their approach.

There are recent standardization efforts such as IEEE 802.11Qbv [36] which aim to codify best practices for provisioning QoS using Ethernet. These approaches focus entirely on meeting guarantees and do not attempt to optimize link bandwidth. However, the global view of the network provided by the SDN architecture allows us to optimize path layouts by formulating it as an MCP problem.

There are prior attempts at provisioning SDN with worst-case delay and bandwidth guarantees. Azodolmolky *et al.* proposed a NC-based model [37] for a single SDN switch that provides an upper bound on delays experienced by packets as they cross through the switch. Guck *et al.* used mixed integer program (MIP) based formulation [38] for provisioning end-to-end flows with delay guarantees, however their approach does not optimize the bandwidth allocation to each queue used by the end-to-end flows at an individual switch. There are approaches [39], [40] that have used queues to rate-limit network traffic and improve end-to-end delay for cloud applications (*e.g.,* MapReduce). However, they do not try to meet a specific end-to-end delay deadline for a given flow, rather accumulate all traffic belonging to a given tenant VM and apply the queue constraints on the host level.

A QoS-enabled management framework for SDN using flow priorities and queueing mechanism was proposed by Xu *et al.* [41]. However, their approach requires discretization of flow delays delays. A QoS routing model was developed in literature [33] that re-configures existing paths and calculates new paths based on the global view and bandwidth guarantees. However, the model requires that the end-to-end delay for a flow is less than or equal to minimum separation times between two consecutive messages, thus limiting its applicability.

Avionics full-duplex switched Ethernet (AFDX) [7]–[9] is a deterministic data network developed by Airbus for safety critical applications. The switches in AFDX architecture are interconnected using full duplex links, and static paths with predefined flows that pass through network are set up. Though such solutions aim to provide deterministic QoS guarantees through static routing, reservation and isolation, they impose several limitations on optimizing the path layouts and on different traffic flows. There have been studies towards evaluating the upper bound on the end-to-end delays in AFDX networks [9]. The evaluation seems to depend on the AFDX parameters though.

There are several protocols proposed in automotive communication networks such as controller area network (CAN) [10] and FlexRay [42]. These protocols are designed to provide strong real-time guarantees but have limitations in how to extend it to varied network lengths, different traffic flows and complex network topologies. With SDN architectures and a flexible QoS framework proposed in this paper, one could easily configure COTS components and meet QoS guarantees with optimized path layouts.

Heine *et al.* proposed a design and built a real-time middleware system, CONES (COnverged NEtworks for SCADA) [43] that enables the communication of data/information in SCADA applications over single physical integrated networks. However, the authors did not explore the synthesis of rules or path optimizations based on bandwidth-delay requirements – all of which are carried out by our system. Qian *et al.* implemented a hybrid EDF packet scheduler [15] for real-time distributed systems. The authors proposed a proportional bandwidth sharing strategy based on number of tasks on a node and duration of these task, due to partial information of the network. In contrast, the SDN controller has a global view of the network, thus allowing for more flexibility to synthesize and layouts the paths and more control on the traffic.

The problem of end-to-end delay bounding in RTS is addressed in literature [44]. The authors choose avionics systems composed of end devices, and perform timing analysis of the delays introduced by end points and the switches. However, the proposed approach requires modification to the switches. Besides the authors do not consider the bandwidth limitations, variable number of flows and flow classifications.

There is a lot of work in the field of traditional real-time networking (too many to enumerate here) but the focus on SDN is what differentiates our work.

X. Conclusion

With the proliferation of commercial-off-the-shelf (COTS) components, designers are exploring new ways of using them, even in critical systems (such as RTS). Hence, there is a need to understand the inherent trade-offs (less customization) and advantages (lower cost, scalability, better support and more choices) of using COTS components in the design of such systems. In this paper, we presented mechanisms that provide end-to-end delays for critical traffic in real-time systems using COTS SDN switches. Hence, future RTS can be better managed, less complex (fewer network components to deal with) and more cost effective.

References

[1] 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.

[2] D. Levin, M. Canini, S. Schmid, and A. Feldmann, "Incremental sdn deployment in enterprise networks," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 473–474.

[3] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[4] J. Spencer, O. Worthington, R. Hancock, and E. Hepworth, "Towards a tactical software defined network," in *Military Communications and Information Systems (ICMCIS), 2016 International Conference on*. IEEE, 2016, pp. 1–7.

[5] T. Pfeiffenberger and J. L. Du, "Evaluation of software-defined networking for power systems," in *Intelligent Energy and Power Systems (IEPS), 2014 IEEE International Conference on*. IEEE, 2014, pp. 181–185.

[6] A. Aydeger, K. Akkaya, M. H. Cintuglu, A. S. Uluagac, and O. Mohammed, "Software defined networking for resilient communications in Smart Grid active distribution networks," in *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–6.

[7] "ARINC Specification 664, Part 7, Aircraft Data Network, Avionics Full Duplex Switched Ethernet (AFDX) Network," 2003.

[8] I. Land and J. Elliott, "Architecting arinc 664, part 7 (afdx) solutions." XILINX, 2009.

[9] H. Charara, J. L. Scharbarg, J. Ermont, and C. Fraboul, "Methods for bounding end-to-end delays on an AFDX network," in *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, 2006, pp. 10 pp.–202.

[10] N. Instruments, "Controller Area Network (CAN) Overview." [Online]. Available: http://www.ni.com/white-paper/2732/en/

[11] Z. Li, Q. Li, L. Zhao, and H. Xiong, "Openflow channel deployment algorithm for software-defined afdx," in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*. IEEE, 2014, pp. 4A6–1.

[12] C. M. Fuchs, "The evolution of avionics networks from ARINC 429 to AFDX," *Innovative Internet Technologies and Mobile Communications (IITM), and Aerospace Networks (AN)*, vol. 65, 2012.

[13] O. S. Specification-Version, "1.4. 0," 2013.

[14] P. Heise, F. Geyer, and R. Obermaisser, "Deterministic openflow: Performance evaluation of SDN hardware for avionic networks," in *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE, 2015, pp. 372–377.

[15] T. Qian, F. Mueller, and Y. Xin, "Hybrid EDF Packet Scheduling for Real-Time Distributed Systems," in *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015, pp. 37–46.

[16] S. Oh, J. Lee, K. Lee, and I. Shin, "RT-SDN: Adaptive Routing and Priority Ordering for Software-Defined Real-Time Networking," 2015.

[17] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, "Dependable end-to-end delay constraints for real-time systems using SDN," *International Workshop on Real-Time Networks (RTN)*, 2017.

[18] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 4, pp. 1270–1283, 2009.

[19] 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*. ACM, 2010, p. 19.

[20] "Pica8 datasheet," http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3297.pdf, [Online].

[21] "Raspberry Pi 3 Model B," https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[22] "The netperf homepage," http://www.netperf.org/netperf/.

[23] J. M. Jaffe, "Algorithms for finding paths with multiple constraints," *Networks*, vol. 14, no. 1, pp. 95–116, 1984.

[24] S. Chen and K. Nahrstedt, "On finding multi-constrained paths," in *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, vol. 2. IEEE, 1998, pp. 874–879.

[25] "Ryu controller," http://osrg.github.io/ryu/, accessed: 2014-11-01.

[26] "IEEE Standard for Ethernet," *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)*, pp. 1–3747, Dec 2012.

[27] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12.

[28] R. L. S. De Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, 2014, pp. 1–6.

[29] M. Erel, E. Teoman, Y. Özçevik, G. Seçinti, and B. Canberk, "Scalability analysis and flow admission control in mininet-based sdn environment," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 18–19.

[30] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos, "Sddc: A software defined datacenter experimental framework," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 189–194.

[31] E. Jo, D. Pan, J. Liu, and L. Butler, "A simulation and emulation study of sdn-based multipath routing for fat-tree data center networks," in *Proceedings of the 2014 Winter Simulation Conference*. IEEE Press, 2014, pp. 3072–3083.

[32] "Production quality, multilayer open virtual switch," http://openvswitch.org/.

[33] S. Oh, J. Lee, K. Lee, and I. Shin, "RT-SDN: Adaptive Routing and Priority Ordering for Software-Defined Real-Time Networking," Tech. Rep., https://cs.kaist.ac.kr/upload_files/report/1406868936.pdf [Online].

[34] P. Ramanathan, "Overload management in real-time control applications using (m, k)-firm guarantee," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 549–559, 1999.

[35] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer Science & Business Media, 2001, vol. 2050.

[36] IEEE. (2015) Ieee 802.11qbv standard. [Online]. Available: http://www.ieee802.org/1/files/private/bv-drafts/d3/802-1Qbv-d3-1.pdf

[37] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *Global Communications Conference (GLOBECOM), 2013 IEEE*. IEEE, 2013, pp. 1397–1402.

[38] J. W. Guck and W. Kellerer, "Achieving end-to-end real-time quality of service with software defined networking," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 70–76.

[39] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *NSDI*, 2015, pp. 1–14.

[40] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 435–448, 2015.

[41] C. Xu, B. Chen, and H. Qian, "Quality of service guaranteed resource management dynamically in software defined network," *Journal of Communications*, vol. 10, no. 11, pp. 843–850, 2015.

[42] "FlexRay Automotive Communication Bus Overview." [Online]. Available: http://www.ni.com/white-paper/3352/en/

[43] E. Heine, H. Khurana, and T. Yardley, "Exploring convergence for SCADA Networks," in *ISGT 2011*, Jan 2011, pp. 1–8.

[44] D. Jin, J. Ryu, J. Park, J. Lee, H. Shin, and K. Kang, "Bounding end-to-end delay for real-time environmental monitoring in avionic systems," in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, March 2013, pp. 132–137.

[45] "Calculating the propagation delay of coaxial cable," https://cdn.shopify.com/s/files/1/0986/4308/files/Cable-Delay-FAQ.pdf, [Online].

Appendix A
Delay Calculations

Remember that some of the critical pieces of information that is required for any such scheme (for ensuring end-to-end delays) is a measure of the delays imposed by the various components in the system. Hence, we need to obtain network delays at each link. We use these estimated delays as the weights of edges of the network graph in the MCP algorithm within the experimental setup to obtain solutions. As discussed earlier, we assume zero queuing delay. The transmission and propagation delays are a function of the physical properties of the network topology. For instance, the transmission delay is calculated as $\frac{\text{packet length}}{\text{bandwidth allocated}}$. For a fixed packet size and specific link bandwidth we consider this as a constant. Besides, the processing delay of an individual switch for a single packet can be empirically obtained. Here we describe our method to obtain upper-bounds on each of these delay components.

Estimation of Propagation Delay

The propagation delay depends on the physical link length and propagation speed in the medium. In the physical media, the speed varies $.59c$ to $.77c$ [45] where c is speed of light in vacuum. We assume that the length of any link in the network to be no more that 100 m. Therefore the propagation delay is upper bounded by $\frac{100\text{m}}{0.66\times3\times10^{6}} = 505$ ns in fiber-link media.

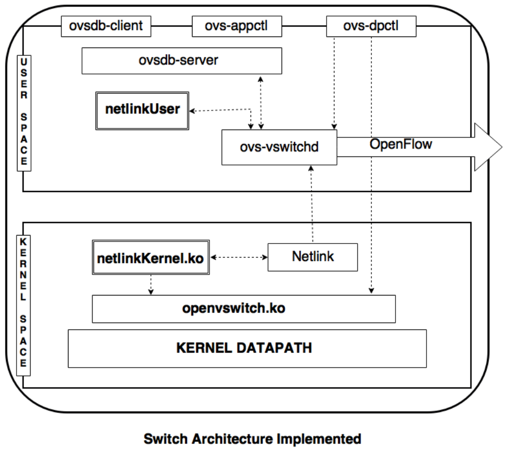


Fig. 7. Interaction of kernel timing module with the existing OVS architecture.

Estimation of Processing Delays

We experimented with a software switch, Open vSwitch (OVS) [32] version 2.5.90 to compute the time it takes to process a packet within its data path. Since this timing information is platform/architecture dependent, we summarized the hardware information of our experimental platform in Table II.

TABLE II
HARDWARE USED IN TIMING EXPERIMENTS

Artifact	Info
Architecture	i686
CPU op-modes	32-bit, 64-bit
Number of CPUs	4
Threads per core	2
Cores per socket	2
CPU family	6
L1d and L1i cache	32K
L2 and L3 cache	256K and 3072K, respectively

We modified the kernel-based OVS data path module called `openvswitch.ko` to measure the time it takes for a packet to move from an ingress port to an egress port. We used `getnstimeofday()` for high-precision measurements. We also developed a kernel module called `netlinkKernel.ko` that copies the shared timing measurement data structure between the two kernel modules and communicates it with a user space program called `netlinkUser`. We disabled scheduler preemptions in the `openvswitch.ko` by using the system calls `get_cpu()` and `put_cpu()`, hence the actual switching of the packets in the data path is not interfered by the asynchronous communication of these measurements by `netlinkKernel.ko`. We also used compilation flags to ensure that `openvswitch.ko` always executes on a specified, separate, processor core of its own (with no interference from any other processes, both from the user space or the operating system). For fairness in the timing measurements and stabilized output, we disabled some of the Linux background processes (*e.g.,* SSH server, X server) and built-in features (*e.g.,* CPU frequency scaling). Figure 7 illustrates the interaction between the modified kernel data path and our user space program.

We used the setup described above with Mininet and Ryu Controller. We evaluated the performance and behavior of OVS data path under different flows, network typologies and packet sizes. We executed several runs of the experiment with UDP traffic with different packet sizes. We observed that average processing time for a single packet within the software switch lies between 3.2 μs to 4.1 μs with average being 3.6 μs and standard deviation being 329.61 ns.

Therefore delay of the edge, *i.e.,* $\mathfrak{D}_k(u,v)$, $\forall(u,v) \in E$ is upper bounded by 3.6+0.505 $\approx$ 4.2 μs. These were the values that were used in the path allocation calculations.

APPENDIX B
QUEUE ASSIGNMENT: MININET OBSERVATIONS

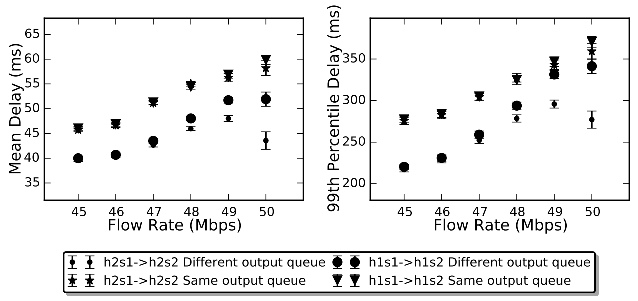


Fig. 8. The mean and 99^{th} percentile per-packet delay for the packets in the active flows in 25 iterations using a two-host four-switch (see Fig. 2(b)) Mininet topology.

We also perform experiments with a two switch, four host topology similar that of presented in Section III-A using Mininet. The purpose of this experiment is to observe the performance impact on software simulations (*e.g.,* Mininet topologies) over the actual ones (hardware switches and ARM hosts). As we can see in Fig. 8 the trends (*e.g.,* isolating flows using separate queues results in lower delays) are similar in both Mininet and hardware experiments – albeit the latencies are higher due to it being a software simulation and also affected by other artifacts (*e.g.,* the experiments are involved in generating traffic on the same machine).