Dependable End-to-End Delay Constraints for Real-Time Systems using SDNs

Rakesh Kumar*, Monowar Hasan†, Smruti Padhy‡, Konstantin Evchenko‡, Lavanya Piramanayagam‡, Sibin Mohan‡ and Rakesh B. Bobba‡

*Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA
†Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
‡National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL, USA
§School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA

Email: {*kumar19, †mhasan11, ‡padhy, ‡evchenk2}@illinois.edu, ‡lav281995@gmail.com,
†sibin@illinois.edu $rakesh.bobba@oregonstate.edu

Abstract

Real-time systems (RTS) require dependable delay guarantees on delivery of network packets. Traditional approaches for providing such guarantees for mission critical applications require the use of expensive custom built hardware and software resources. We propose a novel framework that reduces the management and integration overheads of the traditional approaches by leveraging the capabilities (especially global visibility and management) of Software-defined networking (SDN) architectures. Given the specifications of flows requiring real-time guarantees, our framework synthesizes paths through the network and associated switch configurations that can meet the requisite delay requirements for real-time flows. Using exhaustive simulations as well as emulations with real switch software, we show the feasibility of our approach.

I. INTRODUCTION

Software-defined networking (SDN) [31] has become increasingly popular since it allows for better management of network resources, application of security policies and testing new algorithms and mechanisms. It finds use in a wide variety of domains – from enterprise systems [28] to cloud computing services [22], from military networks [38] to power systems [34] [8]. The centralized (and 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. On the other hand, real-time systems (RTS), especially those with stringent timing constraints, need to reason about delays. Packets must be delivered between hosts with guaranteed upper bounds on end-to-end delays. Examples of such systems include avionics, automobiles, industrial control systems, power substations, manufacturing plants, etc.

Another property of such systems is that they often include traffic flows with mixed criticality, i.e., those with varying degrees of timing (and perhaps even bandwidth and availability) requirements: (a) high priority/criticality traffic that is essential for the correct and safe operation of the system; examples could include sensors for closed loop control and actual control commands in avionics, automotive or power grid systems; (b) medium criticality traffic that is critical to the correct operation of the system, but with some tolerances in delays, packet drops, etc.; for instance, navigation systems in aircraft, system monitoring traffic in power substations, etc.; (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. The high priority flows (that we henceforth refer to as “Class I” traffic) have stringent timing requirements and can often tolerate little to no loss of packets. Often these flows also have predefined priority levels among themselves.

Typically, in many safety-critical RTS, the properties of all Class I flows are well known, i.e., designers will make these available ahead of time. Any changes (addition/removal of flows or modifications to the timing or bandwidth requirements) will often require a serious system redesign. The number (and properties) of other flows could be more dynamic – consider the on-demand video situation 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 each of the aforementioned types 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 surface and vectors. Existing systems, e.g., avionics full-duplex switched Ethernet (AFDX) [5], [17], [25], controller area network (CAN) [20], etc. that are in use in many of these domains are either proprietary, complex, expensive and 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 [29]. In such situations, a dynamic configuration is required to route packets based on

1Systems that, apart from needing logical correctness also imposes temporal correctness requirements – i.e., the system must function correctly, within a given time frame; this time frame is often expressed in the form of a deadline.
switch workloads and flow delays to meet all the high priority QoS (e.g., end-to-end delay) requirements. In addition AFDX protocols require custom hardware [14].

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 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 three separate ones as is the case these days). On the other hand, the current standards used in traditional SDN (OpenFlow [31], [37]) 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 generally less effective [18].

A number of issues arise while developing a software-defined networking infrastructure for use in real-time systems. For instance, flows belonging to Class I need to meet their timing (e.g., end-to-end delay) requirements for the real-time system to function correctly and be dependable. Hence, we need to find a path through the network (and allocate the resources such as bandwidth) that will meet these guarantees. 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 [33], [35], 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 with an understanding of the delays/problems caused by the presence of other flows in that node as well as the network in general.

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

1) We developed mechanisms to guarantee timing constraints for traffic in hard real-time systems that can easily be integrated with COTS SDN hardware (Sections III, IV and V).
2) We illustrate the requirements for isolating flows into different queues to provide stable quality of experience in terms of end-to-end delays (Section III-A) even in the presence of other types of traffic in the system.

We evaluate the effectiveness of the proposed approach with various custom topology and UDP traffic (Section VII).

II. BACKGROUND

1) The Software Defined Networking Model: The communication network orchestrates packet delivery among all network devices by using packet headers. Its configuration drives the two components of the networking software: control and data planes. The control plane decides what packet forwarding and header transformations need to occur at network devices, while the data plane performs the actual actions on packets. In the traditional networking architecture, control and data planes coexist on individual network devices. However, the SDN architecture simplifies access to the network configuration by logically centralizing the control-plane state in a controller. This programmable and centralized state then drives the network devices that perform homogeneous forwarding plane functions [10] and can be modified to control the behavior of the SDN with flexibility.

![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 synthesizes flow rules by using the northbound API.](image)

Current SDN implementations reason about bandwidth instead of delays. Hence, we must find a way to extend the SDN infrastructure to reason about delays for use in real-time systems.

We will work on integrating the other types of traffic in future work.
In order to form the logically centralized state of the SDN, the controller uses the management ports to construct current topology of the SDN and gather data-plane state from each switch. It then makes this state available through a northbound API to be used by the applications. An application (such as our prototype proposed in this paper) uses this API to access a snapshot of SDN state and constructs a model to predict the behavior of the SDN. This state includes the network topology.

2) The Switch: An SDN switch contains a table processing pipeline and a collection of physical ports. Packets arrive at one of the ports, and are processed by the pipeline comprised of one or more flow tables. Each flow table contains flow rules ordered by their priority. Each flow rule is an atomic unit of decision-making, which take the form of actions. During the processing of a single packet, these actions can modify the packet, forward it out of the switch, or drop it.

A packet always matches the highest priority flow rule in any flow table. A packet arriving at the switch is first matched against the rules in the first table and assigned a set of actions to be applied at the end of table processing pipeline. The instructions in a matching rule in any table can choose to manipulate this set and can also apply some actions to the packet before it is processed by the next table in the pipeline. Each flow rule has the following two parts:

- **Match:** A set of packet header field values that apply to the given rule. Some are characterized by single values (e.g., VLAN ID: 1, or TCP 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:** The control operations performed by the flow rule to a matched packet. A switch can apply a set of actions on the packet as part of its instructions. The actions can specify the egress port (OutPutPort) for packets matching the packet. Furthermore, in order to make the appropriate allocation of bandwidth for the matching packets, the OpenFlow specification provides two mechanisms:
  - **Queue References:** Every OpenFlow switch is capable of providing isolation to traffic from different flows by enqueuing them on separate queues for packets on the egress port. Each queue has an associated QoS configuration which 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. All meters on a given are stored in a table and meters can be added/deleted by using messages specified in OpenFlow specification. Each meter has an associated metering rate. Each flow rule can refer to 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 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 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 \(v_p\) 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 \(P_k\). The problem then, is to synthesize flow rules that use queues at each edge \((u, v) \in P_k\) 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 \(\forall (u, v) \in P_k\) the following constraints need to be satisfied:

\[
\sum_{(u,v) \in P_k} d_{f_k}(u, v) \leq D_k, \quad \forall f_k \in F \tag{1}
\]

\[
b_{f_k}(u, v) \geq B_k, \quad \forall (u,v) \in P_k, \forall f_k \in F. \tag{2}
\]

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 4 and 5.

\footnote{We use the terms edge and link interchangeably throughout the paper.}
• **Level 2**: Mapping the path layouts from using Level 1 on to the network topology by using the mechanisms available in OpenFlow. We describe details of our approach in Section VI.

Once we obtain path layouts that satisfy the above constraints, the next step is to map flows assigned to a port to the queues at the port. The possible approaches could be: (a) allocate each flow to an individual queue or (b) multiplex flows in different queues and dispatch the packets based on priority. 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. We carried out some simple 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 propose synthesizing configurations for Class I traffic such that it ensures *complete isolation of packets in a designated class I flow*. 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 setup an experiment using Mininet [26]. The experiment uses a simple topology that contains two switches (s1, s2) connected via a single link as shown in Figure 2(a). Each switch has two hosts connected to it. We configured flow rules and queues in the switches to enable connectivity among hosts at one switch with the hosts at another 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. Finally, we used *ingress policing* such that if traffic from a host exceeds its maximum rate (50 Mbps in this case), its traffic is throttled before it enters the switch.

After configuring the flow rules and queues, we triggered packet flows using netperf [7]: the first starting at the host h1s1 destined to host h1s2 and the second starting at host h2s1 with a destination to host h2s2. Both packet flows are triggered simultaneously and they last for 10 seconds. We changed the rate at which the traffic is sent across both flows and performed each experiment 50 times to measure the average and 99th percentile per-packet delays. Figure 2(b) plots the average value and standard error over all iterations. The x-axis indicates the rate at which the traffic is sent via netperf, while the y-axis shows the mean and 99th percentile delay. The following key observations stand out:

1) The per-packet average and 99th percentile 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 average and 99th percentile delay when flow rates approach the maximum rates. 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 flow to be 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. The maximum delay along a single link can be measured. Such measurements can then be 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 the end-to-end delays for critical real-time flows) and also an overview of our solution.

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

\[
\mathcal{D}_k(P_k) = \sum_{(u,v) \in P_k} \mathcal{D}(u,v).
\]  

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

\[
\mathcal{D}_k(P_k) \leq D_k. \tag{4}
\]

Note that the end-to-end delay for a flow over a path has following delay components: (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 VII-C we use separate queues for each flow with assigned required rates. We also 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 Section VII-C.

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:

\[
\mathcal{B}_k(u,v) = \frac{B_k}{B_e(u,v)} \tag{5}
\]

where \( B_k \) is the bandwidth requirement of \( f_k \) and \( B_e(u,v) \) is total bandwidth of an edge \((u,v) \in E\). Therefore, bandwidth utilization over a path \((P_k)\), for a flow \( f_k \) is defined as:

\[
\mathcal{B}_k(P_k) = \sum_{(u,v) \in P_k} \mathcal{B}_k(u,v). \tag{6}
\]

Note that the bandwidth utilization over a path \( P_k \) for flow \( f_k \) is bounded by

\[
\min_{(u,v) \in E} \mathcal{B}_k(u,v) |V| < \mathcal{B}_k(P_k) \leq \max_{(u,v) \in E} \mathcal{B}_k(u,v) |V|. \tag{7}
\]

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

\[
\mathcal{B}_k(P_k) \leq \hat{B}_k \tag{8}
\]

where \( \hat{B}_k = \max_{(u,v) \in E} \mathcal{B}_k(u,v) |V| \).

**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 \([27]\).

Therefore we consider a polynomial-time heuristic similar to that presented in literature \([12]\). 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 \([12]\). 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

\[
\mathcal{D}_k(u,v) = \frac{X_k \cdot \mathcal{D}(u,v)}{D_k} \tag{9}
\]

\[
\mathcal{B}_k(u,v) = \frac{X_k \cdot \mathcal{B}_k(u,v)}{B_k} \tag{10}
\]

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

\[
\mathcal{B}_k(P_k) \leq X_k. \tag{11}
\]

Besides, the solution to this relaxed problem will also be a solution to the original MCP \([12]\). Likewise, if we relax the delay constraint, Eq. (4) can be rewritten as

\[
\mathcal{D}_k(P_k) = \sum_{(u,v) \in P_k} \mathcal{D}_k(u,v) \leq X_k. \tag{12}
\]
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 \( P_k \) from \( s_k \) to \( t_k \)) if and only if both of the constraints 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., \( D_k(P_k) \leq D_k \) and \( \mathfrak{B}_k(P_k) \leq X_k \) are satisfied if and only if
  \[
d_k[t,i] \leq D_k, \quad \exists i \in [0,X_k] \wedge i \in \mathbb{Z}.
  \]
- **When the delay constraint is relaxed:** The (relaxed) delay and bandwidth constraints, e.g., \( \mathfrak{D}_k(P_k) = \sum_{(u,v) \in P_k} \mathfrak{D}_k(u,v) \leq X_k \) and \( \mathfrak{B}_k(P_k) \leq \hat{B}_k \) are satisfied if and only if
  \[
d_k[t,i] \leq X_k, \quad \exists i \in [0,\hat{B}_k] \wedge i \in \mathbb{Z}.
  \]

V. **Algorithm Development**

A. **Path Layout**

Our proposed approach is based on a polynomial-time solution to the MCP problem presented in literature [12]. Let us consider MCP_HEURISTIC\((N, s, t, W_1, W_2, C_1, C_2)\) is an instance of polynomial-time heuristic solution to the MCP problem that finds a path \( P \) from \( s \) to \( t \) in any network \( N \), such that the constraints \( W_1(P) \leq C_1 \) and \( W_2(P) \leq C_2 \) are satisfied.

The heuristic solution of MCP problem, as summarized in Algorithm 1 works as follows. Let
\[
\Delta(v,i) = \min_{P \in P(v,i)} W_1(P)
\]
(13)
where \( P(v,i) = \{P \mid W_2(P) = i, P \text{ is any path from } s \text{ to } t\} \) is the smallest \( W_1(P) \) of those paths from \( s \) to \( v \) for which \( W_2(P) = C_2 \). Similar to [12], 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 \( W_1(P) \). 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). 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 [12]
\[
\exists i \in [0, \cdots, C_2], \quad d[t,i] \leq C_1.
\]
(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 \( v \) on the path (Line 13). Therefore, the path can be recovered by tracking \( \pi \) starting from destination \( t \) through all immediate nodes until reaching the source \( s \).

Based on this MCP abstraction, we propose 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 (i.e., using Eq. [10]), and solve MCP_HEURISTIC\((N, s_k, t_k, \mathfrak{D}_k, \mathfrak{B}_k, D_k, X_k)\) (Line 3) using Algorithm 1. If there exists a solution, the corresponding path \( P_k \) is assigned for \( f_k \) (Line 6). However, if relaxing bandwidth constraint is unable to return a path, we further relax delay constraint (using Eq. [9]) keeping bandwidth constraint unmodified and solve MCP_HEURISTIC\((N, s_k, t_k, \mathfrak{D}_k, \mathfrak{B}_k, X_k, \hat{B}_k)\) (Line 9). If the path is not found after both of the relaxation steps, the algorithm returns False (Line 41) 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 there exists a solution, 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 \( O(C_2(V E + |\mathcal{P}|)) = O(C_2 V E) \). 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 \( O((X_k + \hat{B}_k)V E) \).

VI. **IMPLEMENTATION**

We implement our prototype as an application that uses the northbound API for the Ryu controller [4]. 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 this section, we describe how we realize a given flow \( f_k \) in the SDN.

---

3Line 3-17 in Algorithm 1 is similar to that of presented in earlier work [12 Sec. 2.2] and for the purposes of this work, we have extended this algorithm for our formulation.
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 intents that are required to express actions that the network needs to perform (for packets for a flow) is the same as the number of switches on the flow path. Each intent is a tuple given by $(\text{Match}, \text{InputPort}, \text{OutputPort}, \text{Rate})$. 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 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 need to 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 precious resources and not all switch implementations provide both of them. As mentioned earlier (Section III), there exist two alternative strategies for allocating bandwidth to individual intents: (a) individual queue/meter for each flow and (b) multiplexing meters and queues across multiple flows. The former strategy limits the rate of the queue/meter to $B_k$. It also places limits on the number of flows that a given switch can support at any point in time. On the other hand, as seen in Section III-A, it guarantees much better isolation among flows. The second
The parameters used in the experiments are summarized in Table I.

Effectiveness of our end-to-end delay guaranteeing mechanisms even in the presence of other traffic in the network (Section VII-A) and 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.

**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 $\mathcal{D}_k = [\mathcal{D}_k(u,v)]_{(u,v) \in E}$ and $\mathcal{B}_k = [\mathcal{B}_k(u,v)]_{(u,v) \in E}$, $\tilde{\mathcal{D}}_k = [\tilde{\mathcal{D}}_k(u,v)]_{(u,v) \in E}$, $\tilde{\mathcal{B}}_k = [\tilde{\mathcal{B}}_k(u,v)]_{(u,v) \in E}$, for each flow $f_k \in F$, respectively, and the delay and bandwidth bounds $D_k \in \mathbb{R}^+$ and $B_k \in \mathbb{R}^+$, respectively, and positive constant $X_k \in \mathbb{Z}$, $\forall f_k \in F$.

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

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

![Fig. 3](image.png)

In this section, we evaluate our proposed solutions using the following methods: (a) an exploration of 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.
| Artifact/Parameter                  | Values                  |
|-----------------------------------|-------------------------|
| Number of switches                | 5                       |
| Bandwidth of links                | 10 Mbps                 |
| Link delay                        | [25, 125] μs            |
| Number of real-time flows         | [1, 5]                  |
| 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 |

Table I: Experimental Platform and Parameters

Fig. 4. Schedulability of the flows in different network topology. For each of the (delay-requirement, number-of-flows) pair (e.g., x and y-axis of the figure), we randomly generate 250 different topology. In other words, total $5 \times 5 \times 250 = 6250$ different topology were tested in the experiments.

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 $f_k \in [1, 5]$ 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 [6]). and based on the discussion on Section VII-C link delays are randomly generated within [25, 125] 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 topology (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 $D_{\text{min}} = \beta \delta_i \mu s$ and increment by $\frac{D_{\text{min}}}{10}$ for each of the remaining flows where $\delta_i$ 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, $\beta = \frac{D_{\text{min}}}{\delta_i}$ and $D_{\text{min}}$ represents x-axis values of Fig. 4. 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 300 μs for a set of 5 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.

Remember our “delay-monotonic” priority assignment where flows with lower end-to-end delays have higher priority.
B. Experiment with Mininet Topology: Demonstrating that the End-to-End Delay Mechanisms Work

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 the proposed scheme using Mininet [26] (version 2.2.1) where switches are configured using Open vSwitch [3] (version 2.3.0). We use Ryu [4] (version 4.7) as our SDN controller. For each of the experiments we randomly generate a Mininet topology using the parameters described in Section VII-A. 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.\(^7\) 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. Our flow rules isolate the non-critical flows from real-time flows. 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. All the experiments are performed in an Intel Xeon 2.40 GHz CPU and Linux kernel version 3.13.0 – 100.

We use netperf (version 2.7.0)\(^7\) to generate the UDP traffic\(^\dagger\) between the source and destination for any flow $f_k$. Once the flow rules and queues are configured, we triggered packets starting at the source $s_k$ destined to host $t_k$ for each of the flows $f_k$. The packets are sent at a burst of 5 with 1 ms inter burst time. All packet flows are triggered simultaneously and last for 10 seconds.

For each set of flows $f_k \in \{2, 5\}$, we randomly generate 25 different network topology. We assume flows are indexed based on priority, i.e., $D_1 < D_2 < \cdots < D_{|F|}$. We set $D_1 = 100\delta_1 \mu$s and increment with 10 for each of the flow $f_k \in F, k > 1$ where $\delta_1$ 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. 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.

Experience and Evaluation: In Fig. 5 we observe the impact of number of flows on the delay. Experimental results are illustrated for the schedulable flows (viz., the set of flows for which both delay and bandwidth constraints are satisfied).

The y-axis of Fig. 5 represents the empirical CDF of average (Fig. 5(a)), 99\textsuperscript{th} percentile (Fig. 5(b)) and worst-case (Fig. 5(c)) round-trip delay experienced by any flow. The empirical CDF is defined as $G_{\alpha}(j) = \frac{1}{\alpha} \sum_{i=1}^{\alpha} \mathbb{1}_{[G_i \leq j]}$, where $\alpha$ is the total number of experimental observations, $G_i$ round-trip delay the $i$-th experimental observation, and $j$ represents the $x$-axis values (viz., round-trip delay) in Fig. 5. The indicator function $\mathbb{1}_{[\cdot]}$ outputs 1 if the condition $[\cdot]$ is satisfied and 0 otherwise.

From our experiments we find that the non-critical flows do not affect the delay experienced by the real-time flows and the average as well as the 99\textsuperscript{th} percentile delay experienced by the real-time flows always meet their delay requirements. This is because our flow rules and queue configurations isolate the real-time flows from the non-critical traffic to ensure that the end-to-end delay requirements are satisfied. 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 $\mathcal{D}_i(u, v) \times \delta_1$ and bounded by $[25\delta_1, 125\delta_1]$ where $\mathcal{D}_i(u, v) \in [25, 125]$ is the delay between the link $(u, v)$ in $i$-th network realization. As seen in Figs. 5(a) and 5(b) the average and 99\textsuperscript{th} percentile round-trip delay are significantly less than the minimum expected round-trip delay bound (e.g., $2 \times 25 \times 4 = 200 \mu$s). This also validates the effectiveness of Algorithm 2. Besides, as seen in Fig. 5 the worst-case delay is also less than the maximum expected delay bound (e.g., 1000 $\mu$s) with probability 1.

In Fig. 6 we illustrate the 99\textsuperscript{th} percentile round-trip delay (represents the y-axis in the figure) with different number of flows (x-axis). As shown in Figures 5 and 6, increasing the number of flows 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.

For our experiments with Mininet and netperf generated traffic, we do not observe any instance for which a set of schedulable flow misses its deadline (i.e., packets arriving after the passing of their end-to-end delay requirements). Thus, based on our empirical results and the constraints provided to the path layout algorithm, we can assert that the schedulable real-time flows will meet their corresponding end-to-end delay requirements.

C. 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

\(^7\)In real systems, the bandwidths allocation would be overprovisioned (as mentioned earlier), our evaluation takes a conservative approach.

\(^\dagger\)Remember that most hard real-time systems use UDP traffic [14, 33].
delays are a function of the physical properties of the network topology. However, 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 transmission delay is calculated as $\frac{\text{packet length}}{\text{bandwidth allocated}}$. In our experiments we assume the packet length is [25,125] bytes and the maximum bandwidth can be allocated allocated in a specific link is 10 Mbps. Then transmission delay on that link will be upper bounded by $\frac{125 \times 8 \text{ bits}}{10 \text{ Mbps}} = 100 \mu s$. Therefore delay of the edge, i.e., $D_k(u,v)$, $\forall (u,v) \in E$ is upper bounded by $3.6 + 0.505 + 100 \approx 105 \mu s$.

Estimation of Transmission 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 $.77c$ 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 $\frac{100m}{0.66 \times 3 \times 10^9} = 505$
ns in fiber-link media.

**Estimation of Processing Delays:** We experimented with a software switch, Open vSwitch (OVS) 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.

| 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 (100, 1000, 1600 bytes). 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. These were the values that were used in the path allocation calculations.

**VIII. DISCUSSION**

Despite the fact that we provide an initial approach to leverage the benefits of the SDN architecture to guarantee guarantee end-to-end delay in safety-critical hard RTS, our proposed scheme has some limitations and can be extended in several
directions. To start with, we intend to validate our implementations in actual hardware switches with more complex topology and analyze the worst-case latency experienced by the flows. Furthermore, most hardware switches limit the maximum number of individual queues that can be allocated to flows. Our current intent realization mechanism reserves one queue per port for each Class I flow. This leads to depletion of available queues. Hence, we need smarter methods to multiplex Class I flows through limited resources and yet meet their timing requirements. Our future work will focus on developing sophisticated schemes for ingress/egress filtering at each RT-SDN-enabled switch. This will also help us better identify the properties of each flow (priority, class, delay, etc.) and then develop scheduling algorithms to meet their requirements.

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 to share 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 [36] 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 for the traffic flows. Results from the network calculus (NC) [27] 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 [12] for solving MCP. We model our delay and bandwidth constraints based on their approach.

In traditional networks, provisioning for real-time applications is hampered by a distributed control-plane. The routers function as a distributed system and eventually reach consensus about a given application’s packets. This consensus requires special protocols such as RSVP [41]. Given that added layer of complexity, Integrated Services IntServ [13] architecture was proposed in the mid 1990s for provisioning networks for real-time applications. Our approach of explicit resource allocation based on requirements of the application lines up with the principles behind IntServ. Finally, recent standardization efforts such as IEEE 802.11Qbv [19] 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 by the SDN architecture allows us to optimize path layouts by formulating it as an MCP problem.

There have been some prior attempts at provisioning SDNs with worst-case delay and bandwidth guarantees. Azodolmolky et al. proposed a NC-based model [9] 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 [16] for provisioning end-to-end flows that provide delay guarantees – they do not provide a solution of what traffic arrival rate to allocate for queues on individual switches for a given end-to-end flow. A QoS-enabled management framework to allow end-to-end communication over SDN is proposed in literature [40]. It classifies flows into two levels, i.e., QoS flow and best-effort. It uses flow priority and queue mechanism to obtain QoS control to satisfy the requirement.

It developed a scalable routing scheme that re-configures existing paths and calculates new paths based on the global view and bandwidth guarantees. A priority ordering scheme is also presented for flows sharing the same switch and to avoid contention. However, the basic requirement of the model used in that work: i.e., end-to-end delay being less than or equal to minimum separation times between two consecutive messages, limits applicability of their scheme for a wide range of applications.

Avionics full-duplex switched Ethernet (AFDX) [5], [11], [25] is a deterministic data network developed by Airbus for safety critical applications and is being used in aircraft such as Airbus A380. The switches in AFDX architecture are interconnected using full duplex links and static paths with predefined flows that passes through network are set up. Though such solutions aim to guarantee deterministic QoS through static routing, reservation and isolation, it imposes several limitations on optimizing the path layouts and on different traffic flows. 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 and backup paths. There have been studies towards evaluating the upper bound on the end-to-end delays in AFDX networks [11]. The evaluation seems to depend on the AFDX parameters though.

Several studies investigated the delay introduced by a scheduling algorithm implemented in the switch to route the packets through the switching fabric [15], [24], [39]. The earlier work indicates that deriving processing delays is not trivial for certain common scheduling algorithms such as iSLIP [30]. Hence, alternative algorithms that are more amenable to bounding the delay were proposed. Delay bounds in switching devices can not provide real-time guarantees for the whole network per se,
but they must be considered when designing such network. The authors [15] build on this knowledge and propose an algorithm to construct close-to-optimal network of switches that meets real-time requirements. Our approach is different in a sense that we consider existing network topology that can be arbitrarily complex.

There are several protocols proposed in automotive communication networks such as controller area network (CAN) [20] and FlexRay [2]. CAN [20] is the international standard for in-vehicle communication in the automotive industry. It is peer-to-peer network and transmits a CAN frame when the bus is not busy and if multiple nodes try to transmit, the node with highest priority gets the access. FlexRay [2] uses TDMA scheme to manage multiple nodes and allows complex network topology. 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.

Heine et al. proposed a design and built a real-time middleware system, CONES (COnverged NEtworks for SCADA) [17] that enables the communication of data/information in SCADA applications over single physical integrated networks. These solutions leveraged some of the existing IP technologies such as Multiprotocol Label Switching (MPLS) and DiffServ for network resource reservation/management, and are based on iDSRT [32] that uses Earliest deadline first scheduling algorithm (EDF) for scheduling of network traffic and CPU time based on the deadline requirements. 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 [35] for real-time distributed systems. The scheduler transmits messages with a delay introduced by periodic message receiving task that accepts messages from the network buffer and releases to the EDF scheduler queue. This approach makes the system more predictable. At the network level, to reduce network delay variance, the scheduler is integrated with Linux traffic control and is used in combination with hierarchical token bucket (HTB). The network interface with HTB is configured for two queues (viz., real-time and background traffic) and priorities. The authors proposed a proportional bandwidth sharing strategy based on number of tasks on a node and duration of these tasks. Though these strategies were adopted to make the system more predictable, due to partial information of the network, it leaves lots of scope for improvements. 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 (e.g., [23]). The authors choose avionics systems composed of end devices, such as cameras and intermediate switching devices as a motivating example 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 (real-time vs non-real-time). There is a lot of work in the field of traditional real-time networking (too many to enumerate here) but the focus on SDNs 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] Calculating the propagation delay of coaxial cable. [https://cdn.shopify.com/s/files/1/0986/4308/files/Cable-Delay-FAQ.pdf] [Online].
[2] FlexRay Automotive Communication Bus Overview.
[3] Production quality, multilayer open virtual switch. [http://openvswitch.org/]
[4] Ryu controller. [http://osrg.github.io/ryu/] Accessed: 2014-11-01.
[5] ARINC Specification 664, Part 7, Aircraft Data Network. Avionics Full Duplex Switched Ethernet (AFDX) Network. 2003.
[6] IEEE Standard for Ethernet. IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008), pages 1–3747, Dec 2012.
[7] The netperf homepage, 2016.
[8] 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, pages 1–6. IEEE, 2016.
[9] 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, pages 1397–1402. IEEE, 2013.
[10] 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), 17(4):1270–1283, 2009.
[11] 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), pages 10 pp.–202, 2006.
[12] S. Chen and K. Nahrstedt. On finding multi-constrained paths. In Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on, volume 2, pages 874–879. IEEE, 1998.
[13] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In ACM SIGCOMM Computer Communication Review, volume 22, pages 14–26. ACM, 1992.
[14] C. M. Fuchs. The evolution of avionics networks from ARINC 429 to AFDX. *Innovative Internet Technologies and Mobile Communications (IITM), and Aerospace Networks (AN)*, 65, 2012.

[15] S. Gopalakrishnan, M. Caccamo, and L. Sha. Switch scheduling and network design for real-time systems.

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

[17] E. Heine, H. Khurana, and T. Yardley. Exploring convergence for SCADA Networks. In *ISGT 2011*, pages 1–8, Jan 2011.

[18] 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*, pages 372–377. IEEE, 2015.

[19] IEEE. IEEE 802.11qpv standard. 2015.

[20] N. Instruments. Controller Area Network (CAN) Overview.

[21] J. M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[22] R. Jain and S. Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[23] 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*, pages 132–137, March 2013.

[24] K. Kang, K.-J. Park, L. Sha, and Q. Wang. Design of a crossbar voq real-time switch with clock-driven scheduling for a guaranteed delay bound. *Real-Time Systems*, 49(1):117–135, 2013.

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

[26] 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*, page 19. ACM, 2010.

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

[28] D. Levin, M. Canini, S. Schmid, and A. Feldmann. Incremental sdn deployment in enterprise networks. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 473–474. ACM, 2013.

[29] 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)*, pages 4A6–1. IEEE, 2014.

[30] N. McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM Trans. Netw.*, 7(2):188–201, Apr. 1999.

[31] 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*, 38(2):69–74, 2008.

[32] H. Nguyen, R. Rivas, and K. Nahrstedt. idSRT: Integrated Dynamic Soft Real-Time Architecture for Critical Infrastructure Data Delivery over WLAN, pages 185–202. Springer Berlin Heidelberg, 2009.

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

[34] 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*, pages 181–185. IEEE, 2014.

[35] 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)*, pages 37–46, July 2015.

[36] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

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

[38] J. Spencer, O. Worthington, R. Hancock, and E. Hepworth. Towards a tactical software defined network. In *Military Communications and Information Systems (ICMICS), 2016 International Conference on*, pages 1–7. IEEE, 2016.

[39] Q. Wang and S. Gopalakrishnan. Adapting a main-stream internet switch architecture for multihop real-time industrial networks. *IEEE transactions on industrial informatics*, 6(3):393–404, 2010.

[40] C. Xu, B. Chen, and H. Qian. Quality of service guaranteed resource management dynamically in software defined network. *Journal of Communications*, 10(11):843–850, 2015.

[41] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. 46V: A new resource reservation protocol. *IEEE network*, 7(5):8–18, 1993.