MC-SDN: Supporting Mixed-Criticality Real-Time Communication Using Software-Defined Networking

Kilho Lee, Minsu Kim, Taejune Park, Hoon Sung Chwa, *Member, IEEE*, Jinkyu Lee, *Member, IEEE*, Seungwon Shin, and Insik Shin, *Member, IEEE*

***Abstract*—Despite recent advances, there still remain many problems to design *reliable* cyber-physical systems. One of the typical problems is to achieve a seemingly conflicting goal, which is to support timely delivery of real-time flows while improving resource efficiency. Recently, the concept of *mixed-criticality* (MC) has been widely accepted as useful in addressing the goal for real-time resource management. However, it has not been yet studied well for real-time communication. In this paper, we present the first approach to support MC flow scheduling on switched Ethernet networks leveraging an emerging network architecture, software-defined networking (SDN). Though SDN provides flexible and programmatic ways to control packet forwarding and scheduling, it yet raises several challenges to enable real-time MC flow scheduling on SDN, including: 1) how to handle (i.e., drop or re-prioritize) out-of-mode packets in the middle of the network when the criticality mode changes and 2) how the mode change affects end-to-end transmission delays. Addressing such challenges, we develop MC-SDN that supports real-time MC flow scheduling by extending SDN-enabled switches and OpenFlow protocols. It manages and schedules MC packets in different ways depending on the system criticality mode. To this end, we carefully design the mode change protocol that provides analytic mode change delay bound, and then resolve implementation issues for system architecture. For evaluation, we implement a prototype of MC-SDN on top of Open vSwitch, and integrate it into a real world network testbed as well as a 1/10 autonomous vehicle. Our extensive evaluations with the network testbed and vehicle deployment show that MC-SDN supports MC flow scheduling with minimal delays on forwarding rule updates and it brings a significant improvement in safety in a real-world application scenario.**

***Index Terms*—Cyber-physical systems (CPSs), mixed-criticality (MC) scheduling, real-time communication, software-defined networking (SDN).**

I. Introduction

As individual systems (things) get larger and connected with each other, there is a growing demand for designing *reliable* cyber-physical systems (CPSs), which needs to achieve not only functional correctness, but also temporal one. Recent advances in embedded systems and communication technologies have led to the following two critical trends regarding designing reliable CPSs. First, CPSs generally rely on networks that interconnect sensors, controllers, and actuators to achieve the function of real-time sensing and dynamic control, such as vision-based simultaneous localization and mapping (SLAM) in self-driving cars. These networks often face new challenges with increased demands on bandwidth and latency requirements that go beyond the capacity of the standard networks. To address such challenges, many CPS industries, such as automotive and avionics, seek to develop next-generation networks using switched Ethernet [1], [2].

The second important trend is toward *mixed-criticality* (MC) systems that integrate application components with different levels of criticality onto common hardware platforms in order to reduce cost, which is essential in automotive and avionics industries. The scheduling problem of MC systems has been intensively studied in recent years, commonly addressing two seemingly conflicting goals: 1) logical separation between applications with different criticality levels and 2) efficient scheduling of shared resources. A key principle in balancing such conflicting goals is to employ *mode*-based MC scheduling such that the system provides different levels of schedulability guarantee for different system modes. The majority of studies in the literature proposed various scheduling algorithms and analyses (see [3] for a survey), indicating that mode-based MC scheduling helps improve schedulability in the case of processor scheduling. Following this implication, a few studies investigated the scheduling issue of MC flows on various networks including controller area network (CAN) [4] and network-on-chip (NoC) [5]–[8], and the criticality-level management issue

Manuscript received August 28, 2018; revised February 25, 2019; accepted April 24, 2019. Date of publication May 9, 2019; date of current version July 31, 2019. This work was supported in part by the Defense Acquisition Program Administration/Agency for Defense Development (DARPA/ADD) (High-Speed Vehicle Research Center of KAIST) under Grant UD170018CD, in part by the Basic Science Research Program (BSRP) under Grant NRF-2015R1D1A1A01058713, in part by the Engineering Research Center (ERC) under Grant NRF-2018R1A5A1059921, in part by the Institute for Information & Communications Technology Planning & Evaluation (IITP) (Resilient Cyber-Physical Systems Research) under Grant 2014-0-00065, in part by the National Research Foundation (NRF) under Grant 2015M3A9A7067220, Grant 2019R1A2B5B02001794, Grant 2017H1D8A2031628, and Grant 2017M3A9G8084463, and in part by IITP (Global SDN/NFV OpenSource Software Core Module/Function Development) under Grant 2015-0-00575. An earlier version of this paper was presented at the IEEE RTSS 2018 [52]. *(Corresponding author: Jinkyu Lee.)*

K. Lee, M. Kim, T. Park, and I. Shin are with the School of Computing, KAIST, Daejeon 34141, South Korea (e-mail: khlee.cs@kaist.ac.kr; minsu@kaist.ac.kr; taejune.park@kaist.ac.kr; insik.shin@cs.kaist.ac.kr).

H. S. Chwa is with the Information and Communication Engineering, DGIST, Daegu 42988, South Korea (e-mail: chwahs@dgist.ac.kr).

J. Lee is with the Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea (e-mail: jinkyu.lee@skku.edu).

S. Shin is with the School of Electrical Engineering, KAIST, Daejeon 34141, South Korea (e-mail: claude@kaist.ac.kr).

Digital Object Identifier 10.1109/JIOT.2019.2915921

on clock-synchronized switches [9]. However, no solutions are yet presented that enable mode-based different scheduling for MC flows in switched Ethernet networks.

In this paper, motivated by the above trends, we aim to support MC flows on switched Ethernet networks. In particular, we seek to develop mode-based in-network MC flow scheduling, in order to enforce MC scheduling more effectively and to accommodate even legacy flows. However, it is not feasible to achieve it with traditional switches since their static nature cannot support the dynamic behavior of mode-based MC scheduling. As mentioned before, MC scheduling requires a dynamic scheduling policy which takes different actions in different modes. However, traditional switches should use only static scheduling behavior that is determined at design time. One can update the firmware on switches in order to enable new scheduling behavior, but firmware updates typically take long and require reboots. Thereby, it cannot change scheduling behavior at runtime. Recently, IEEE Time Sensitive Network (TSN) standards [10]–[16] have been proposed to support various classes of time-sensitive traffics on Ethernet with bounded latency, low packet delay variation, and low packet loss. Yet, TSN relies on static scheduling and thereby cannot support mode-based dynamic MC flow scheduling.[1]

Here, we propose to leverage *software-defined networking* (SDN) for supporting MC flows on switched Ethernet, taking advantage of its flexible nature. SDN is an emerging network architecture toward a novel control paradigm by separating the roles of network control (i.e., control plane) and packet forwarding function (i.e., data plane). The control function, formerly tightly bounded in individual network devices, is migrated into external software, becoming directly programmable. This new programmatic way of controlling the forwarding function allows network managers to easily update forwarding policies while the network is running. To this end, a software-based SDN controller exchanges control messages with SDN-enabled switches through a standard protocol such as OpenFlow [17], to collect network information or manage forwarding rules in each switch.

Despite the SDN opportunity, leveraging SDN for MC scheduling raises several issues to explore, including detecting and handling mode change. The system mode relies on the behavior of each flow, such as a release interval and the size of each periodic message, but SDN switches cannot be aware of the behavior. In addition, it may impose long and unpredictable delays in conducting mode changes based on the OpenFlow, the de-facto standard SDN protocol. Our motivational benchmark shows that the delay is up to 86 ms with a large variation. Such a delay can be added to the end-to-end delays of high-criticality flows and compromise real-time guarantees for the flows. For instance, an obstacle-detecting camera operating at 60 frames/s in high-criticality mode may miss its end-to-end deadline due to the mode change delay, and such a deadline miss could cause a car crash.

This paper presents a novel SDN-based network system, named MC-SDN, which effectively supports flow monitoring and mode change for MC scheduling. We first perform an empirical analysis of mode change delays and identify three major delay factors: 1) mode change arrangement; 2) new rule update; and 3) out-of-mode packet handling. Leveraging such findings, MC-SDN is designed to completely change the way of mode change, a shift from a controller-driven centralized to a switch-driven distributed approach. MC-SDN switches perform flow behavior monitoring and conduct mode changes with minimal delays. Each switch carries out an efficient update of packet forwarding rules within SDN data plane and rearranges packet queues according to the updated rules. This way, MC-SDN eliminates the causes of major delay factors, including OpenFlow communication with the SDN controller, intraswitch communication, and out-of-mode packet transmission. Thus, MC-SDN not only significantly reduces the mode change delay, but also strictly limits its variation to derive a close upper bound.

In addition to the fast, predictable mode change mechanism, we also present a new mode management protocol that: 1) supports sustainable mode changes not only from a low-criticality mode to a high-criticality one but also in the opposite direction and 2) preserves the system consistency such that every switch operates in the same system mode. Also, we discuss how to extend MC-SDN toward more than two criticality levels.

To evaluate the performance of MC-SDN, we first have derived a worst-case delay bound for a mode change under MC-SDN. We then have implemented a prototype of MC-SDN on top of Open vSwitch (OVS) [18] and evaluated it on a real-world network testbed composed of 29 single board computers. Our extensive evaluation shows that MC-SDN effectively reduces the mode change delay by two orders of magnitude compared to the standard SDN and that the delay stays strictly lower than its upper bound. In addition, we conducted a case study on autonomous driving, where MC-SDN is deployed in a 1/10 scale autonomous car.[2] It shows that MC scheduling powered by MC-SDN helps to improve the safety of the driving car in the real world. Also, we demonstrate a key application of MC-SDN, a fault-recovery system. We made several changes to MC-SDN for the application, and demonstrate that MC-SDN for the application is effective in recovering a link fault in the 1/10 scale autonomous car.

The contributions of this paper are summarized as follows.

1) We analyze the limitations of the standard SDN interface (i.e., OpenFlow) when supporting mode change and identify major delay factors, which is the basis for the design of MC-SDN (Section III).
2) We propose MC-SDN that presents a novel mechanism and addresses its implementation issues to support MC scheduling in switched Ethernet (Sections IV and V). To the best of our knowledge, this is the first work that develops mode-based MC scheduling mechanisms on SDN/OpenFlow networks.
3) We derive a worst-case delay bound for a mode change under MC-SDN (Section VI).

[1] More detailed explanation will be given in Section XII-A.

[2] See our demo video illustrating how MC-SDN supports real-time MC flows for Autonomous Emergency Braking (AEB). [Online]. Available: http://cps.kaist.ac.kr/mcsdn

4) We propose a new mode management protocol for MC-SDN, for sustainable mode changes and system consistency (Section VII).
5) We implement a prototype of MC-SDN and reveal orders of magnitude improvement by MC-SDN in mode change delays (Section VIII).
6) We conduct a case study of a scaled autonomous car and demonstrate the effectiveness of MC-SDN in supporting MC flows (Section IX).
7) We demonstrate a target application of MC-SDN, as well as its effectiveness for link fault resiliency (Section X).
8) We discuss a possible extension of MC-SDN toward multicriticality levels (Section XI).

II. System Model and Background

A. System Model

1) Flow Model: We consider an MC system composed of a set of periodic real-time *flows* on switched Ethernet network. A *flow* is a set of potentially unbounded series of periodic *messages*. The *message* can be divided into multiple *packets*, depending on the maximum transmission unit (MTU) of the link (e.g., 1500 bytes on Ethernet). Each flow has a set of properties and requirements specified by a set of attributes, $<T$, C, D, L, *src*, *dst*, *route*$>$: T is a minimum separation time between two consecutive messages (i.e., period). C is a maximum byte size of each message. D specifies a relative end-to-end deadline of each message (note that $D \leq T$). L denotes a criticality level. *Src* and *dst* specify a source node and a destination node (with IP addresses and port numbers), respectively. *Route* is a sequence of nodes that connects *src* to *dst*.

Since a message may consist of multiple packets, we consider a network system where end nodes annotate message information between the transport layer and the application layer as a shim-header; the header contains *message id*, *message size*, and *sequence number*. The shim-header structure is commonly used for message transmission libraries, such as UDPROS in the robot operating system (ROS) [19], a widely used framework for robot and autonomous driving systems.

2) Priority-Based Flow Scheduling: Each switch in the network stores and forwards messages according to the *priority* of each flow. We consider switches that use a strict priority queue [20], [21], which ensures that high priority packets are forwarded ahead of low priority packets. The priority of each flow is specified by forwarding rules of the switch; it is fixed unless the forwarding rules are changed.

3) Mixed-Criticality System: This paper considers MC systems supporting real-time flows which come with multiple requirements according to the criticality levels; the higher the criticality level is, the more conservative the requirement estimation is. For instance, a flow generated by a camera sensor may change its behavior (i.e., frame rate or size) according to the situations, and it has multiple requirement estimations in each system mode. Hence, the period (or the minimum separation) T and the message size C are now defined as $T(L)$ and $C(L)$, respectively, with the following constraints:

$$L_1 \text{ exhibiting a higher criticality than } L_2 \Rightarrow T(L_1) \leq T(L_2) \text{ and } C(L_1) \geq C(L_2)$$

for any two criticality levels L_1 and L_2 [4].

For simplicity, we focus on dual-criticality systems [4], [6], [7] that have two criticality levels, HI (high) and LO (low), where HI exhibits a higher criticality than LO. The correctness of the dual-criticality system is defined as follows: it should satisfy the LO requirement of all flows when the system is in LO mode, and the HI requirement of every HI flow when the system is in HI mode, which is a typical requirement of MC real-time systems [3]. The system starts in LO mode. If all flows behave with satisfying the LO requirement, the system stays in LO mode. However, if any flow violates the mode-specific requirements (i.e., *mode violation*) by generating messages more frequently or larger than its LO requirement, the system then changes its mode into HI mode (i.e., *mode change*) and each switch should update its forwarding table with HI mode rules to favor HI flows; it may drop LO flows or promote the priority of HI flows.

Beyond the dual-criticality systems, MC-SDN also applicable to the systems having more than two criticality levels, to be discussed in Section XI.

4) Priority Assignment: We assume that the system utilizes *rate monotonic* (RM) [22] priority assignment. The shorter the period, the higher the priority. In HI mode, LO flows could be dropped or changed to have lower priority. Although we consider RM in this paper, thanks to the generality of the proposed system, it also supports any kind of fixed-priority scheduling policy.

B. SDN Background and Opportunity

SDN is a recently devised networking technology; thanks to its flexibility and cost-efficiency, now it is widely adopted in real-world networking environments. Unlike legacy network devices, it decouples its control plane, determining/handling network policies, from the data plane, in charge of carrying network packets, to enable dynamic and flexible network control [23]. The decoupled control plane becomes a software component running on a separate device; therefore, the data plane requires and receives network policies from the remote control plane. In addition, it standardizes an interface between the control plane and the data plane; thus it helps network administrators to focus on managing network policies. The most popular network interface between the control plane and the data plane is *OpenFlow* [17]. It handles traffic by a *flow entry* determined by *match-action* tuple. *Match* contains a set of match fields, such as source/destination IP addresses, to match flow entries with incoming packets, and *action* contains a set of instructions how to handle the matched packets. Each switch has a *forwarding table* which holds flow entries, and handles packets according to the flow entries in the table. Note that we will refer *flow entry* as *rule* in this paper, to avoid confusion.

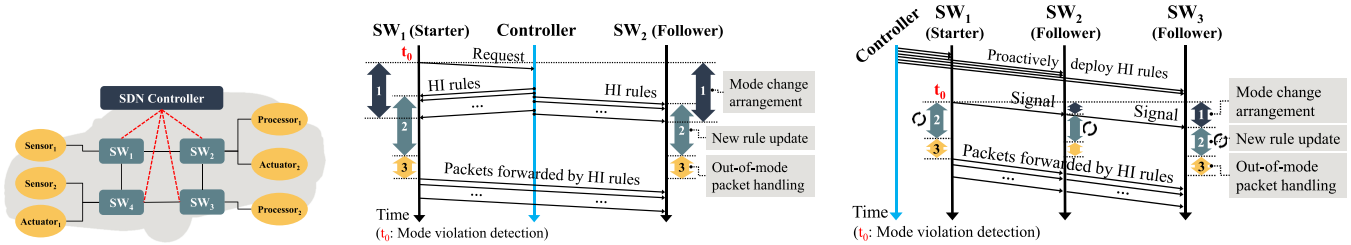


Fig. 1. System architecture and mode change protocol overview. (a) MC networking system overview. (b) Std-SDN: existing controller-driven mode change. (c) MC-SDN: switch-driven mode change.

The decoupled structure of SDN brings high flexibility to network management, and it has a great opportunity to support an MC scheduling which requires highly dynamic packet handling [see Fig. 1(a)]. The main idea of MC scheduling is to apply a differentiated scheduling policy depending on the system mode. For example, a packet should be forwarded in LO mode but dropped in HI mode. Despite the demand for an MC scheduling, a traditional network system, including switched Ethernet, is impossible to support that due to the static nature. Traditional switches only use static scheduling policies determined at design time. To change the policy, a network administrator should update the firmware of the switch by hand. It may take a long delay and require reboots, thus it is impossible to change the policy at runtime. On the other hand, the decoupled structure and the OpenFlow interface of SDN enable switches to dynamically change the policy, even at runtime. The flexibility of SDN could become a key basis to support MC scheduling in switched Ethernet.

III. Challenges of MC Scheduling on SDN

Despite the opportunity from the flexible nature of SDN, it yet raises several challenges to enable real-time MC scheduling on SDN. SDN lacks proper mechanisms for mode-based scheduling, such as mode violation detection and mode change protocols. Furthermore, SDN is originally designed without considering real-time support. In particular, its centralized control paradigm may yield long and unpredictable delays during mode change, which is the most important feature of MC scheduling. Thereby, this section examines significant delay factors in mode change that will be the basis of the MC-SDN design to be proposed.

A. Motivation Experiment: Controller-Driven Mode Change

We conducted benchmark experiments on a real-world network testbed (refer to Section VIII for the testbed details), in order to estimate how long it takes to complete a mode change. For the experiment, we developed a basic controller-driven mode change approach in accordance with the principle of the standard request-response SDN protocol, as follows [see Fig. 1(b)]. A switch sends a mode change request to the SDN controller upon seeing a predefined flag in a packet (i.e., note that we used the flag since a default SDN switch has no way to detect any mode violation), as if it observes a mode violation. Upon receiving the request, the controller deploys new rules to all switches. During the experiments, we measured *mode change delay* as an elapsed time between the time to send a mode change request and the time all the switches are ready to handle packets according to the new HI mode rules. The experiments were performed on a small network, where a switch and end nodes are connected in a star topology. An SDN controller is connected to the switches, and it deploys 30 HI rules in a mode change. Fig. 2 depicts an empirical cumulative distribution function (CDF) of the mode change delay of 100 trials. The figure shows that the mode change delay fluctuates widely and takes as long as 86 ms in the worst case. Such a long delay could damage real-time guarantee of the HI flow. For instance, an obstacle detecting camera which operates at 60 frames/s in HI mode (i.e., period of 16.7 ms) may miss a deadline due to the delay.

TABLE I
Breakdown of Mode Change Delay

	Avg.(Stdev.) (ms)
Overall	50.65 ($\pm$17.94)
Mode change arrangement	31.83 ($\pm$10.54)
New rule update	12.06 ($\pm$0.52)
Out-of-mode packet handling	9.12 ($\pm$8.06)

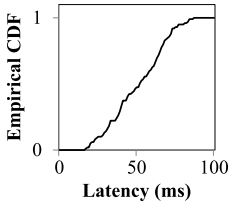


Fig. 2. Empirical CDF of mode change delay.

B. Breakdown of Mode Change Delay

The controller-oriented principle of SDN causes long mode change delays. As shown in Table I, we break the delays down into three parts to closely investigate delay factors: 1) *mode change arrangement*; 2) *new rule update*; and 3) *out-of-mode packet handling* [see Fig. 1(b)]. The remainder of this section elaborates delay factors in each step.

1) Mode Change Arrangement: The mode change arrangement step involves OpenFlow communication between the controller and switches, which is the root cause of the mode switch delay. The mode switch starts when a switch detects HI mode flow behavior and sends a mode change request to the controller. In response to the request, the remote controller sends HI mode rules enclosed in OpenFlow messages to every switch. With the messages, each switch: 1) recognizes a mode change and 2) receives new rules for HI mode. Such controller-switch communication typically causes a significant delay. According to a measurement study [24], OpenFlow communication throughput and latency widely vary depending on the controller's setup and load; for instance, the latency varies from 100 μs to 1268 ms. As shown in Table I, we also observe that the OpenFlow communication delay is long and fluctuated (up to 50 ms) despite our simple benchmark setup. It is very difficult to reduce and bound the delay, since the controller consists of complicated software layers such as an OS network stack, an SDN controller framework, and an SDN controller application. An OpenFlow message passes through those layers and could be delayed by a scheduling policy or an optimization technique (e.g., batching) in each layer.

2) New Rule Update: The rule update step includes communication between switch internals, which incurs significant delays. This step starts when a switch receives new HI rules from the controller and finishes when the switch updates its forwarding table with the new rules received. The main cause of delay in this step lies in the design structure of SDN switch. An SDN switch typically comprises a number of independent modules that follow the principle of modular design for performance and management. For example, the *switch manager* module receives forwarding rules from the SDN controller through OpenFlow communication, and the forwarding rules are transferred to the *datapath* module that conducts packet forwarding according to the rules, which causes non-negligible internal communication delays. Our benchmark experiment results show that it can take up to 14 ms in OVS, which is the de facto standard software switch, for the datapath module (a kernel module) to bring forwarding rules from the switch manager (a user process). It is worthwhile to note that this step can partially overlap with the mode change arrangement step [see Fig. 1(b)]. Yet, the delay of this step is too long to hide in the overlap; our benchmark experiment shows that the average overlap is 2.35 ms and the standard deviation is 0.18 ms. It is very difficult to reduce and bound the delay because the internal communication channel is highly complicated due to optimization techniques, such as asynchronous I/O and batching.

3) Out-of-Mode Packet Handling: The out-of-mode packet handling step also incurs a long delay. When the rule update step has been done, the packets that were enqueued according to LO mode rules could remain in the queue (note that we call them *out-of-mode* packets), and they may delay HI flows. In our benchmark, we observed up to 240 out-of-mode packets (1442 bytes each) when two LO flows and one HI flow share a link; it takes up to 27 ms to transmit them at 100 Mb/s. This delay is very hard to reduce since the link speed depends on the physical constraints.

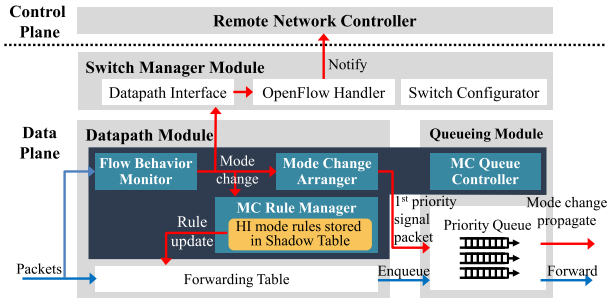


Fig. 3. MC-SDN system architecture.

IV. MC-SDN: System Design

We propose MC-SDN that supports real-time MC scheduling on SDN, addressing the challenges raised to enable mode change properly. It completely shifts the way of conducting mode changes from *controller-driven* to *switch-driven* [i.e., from Fig. 1(b) and (c)] with careful design of data plane components. This new approach not only significantly reduces mode change delays but also strictly limits the delays to predictable upper bounds. Note that, for ease of presentation, we first describe our system design based on the situation of a mode change from LO to HI. We then present our mode management protocol, including a mode change from HI to LO, in Section VII.

As shown in Fig. 1(c), MC-SDN adopts a paradigm of switch-driven mode change, where switches detect mode violation and enable mode change without the controller being involved. To this end, MC-SDN extends SDN data planes with four additional components (see blue components in Fig. 3). *Flow behavior monitor* carries out a new monitoring function to detect flow behavior that violates mode-specific requirements. When a switch detects any mode violation (i.e., HI mode behavior), the switch (called *starter*) triggers a mode change such that its *mode change arranger* notifies this event to all other switches (called *followers*) to enable the system-wide mode change, which reduces communication delays substantially bypassing the SDN controller. After sending a mode switch signal or receiving the signal, *MC rule manager* updates a forwarding table with its own new mode rules stored locally (proactively received HI rules) while minimizing delays in rule updates. The MC rule manager does not only completely eliminate OpenFlow communication, which is a main cause of the mode change delay, but also minimize intraswitch communication, which is another major delay factor. When the forwarding table update is finished, *MC queue controller* rearranges queues based on the new rules to significantly reduce additional delays caused by out-of-mode packets. The rest of this section describes the design principles for each component and how to address the challenges.

A. Flow Behavior Monitor

Beyond a simple byte counter of the current SDN switches, *flow behavior monitor* provides additional monitoring capability of flow behavior such as message sizes and arrival intervals. Even though switches are capable of monitoring such

Algorithm 1 Flow Behavior Monitor

When the message k of the flow τ arrives at $E_{\tau,k}$,

1: $G_{\tau,k}(\mathsf{LO}) \leftarrow max\{G_{\tau,k-1}(\mathsf{LO}), E_{\tau,k-1} - J_\tau\} + T_\tau(\mathsf{LO})$
2: **if** $E_{\tau,k} < G_{\tau,k}(\mathsf{LO})$ **or** $C_{\tau,k} > C_\tau(\mathsf{LO})$ **then**
3: **if** current mode is LO **then**
4: mode change to HI
5: **end if**
6: **end if**

advanced features, yet it is not straightforward to determine whether or not flows behave according to per-mode requirements. As an example, suppose several messages of a flow arrive at a switch more frequently than the LO mode period of the flow. This can happen because the flow actually transmits messages at a faster rate, or because those messages arrive close together due to irregular network congestion. In the former case, it is necessary to change to HI mode. So it is important to understand the situation accurately. To this end, we perform traffic pattern analysis based on the *sporadic invariant* with a guide time [4]. Algorithm 1 presents the mechanism of the flow behavior monitor. For each message k in a flow τ, the monitor checks $C_{\tau,k}$ and $E_{\tau,k}$, where $C_{\tau,k}$ and $E_{\tau,k}$ represent the size and the arrival time of message k, respectively. The monitor also calculates a guide time, $G_{\tau,k}$, which is a bound of an arrival time, $E_{\tau,k}$. Two consecutive messages could be as close as either: 1) LO period—release jitter [i.e., $T_\tau(\mathsf{LO}) - J_\tau$], if the last message arrived later than its guide time or 2) LO period [i.e., $T_\tau(\mathsf{LO})$], if messages are arriving at the maximum rate. Thereby, the guide time is calculated as [4]

$$G_{\tau,k}(\mathsf{LO}) = \max\{G_{\tau,k-1}(\mathsf{LO}), E_{\tau,k-1} - J_\tau\} + T_\tau(\mathsf{LO}).$$

For each message k of flow τ, if $E_{\tau,k} \geq G_{\tau,k}(\mathsf{LO})$, the message k's behavior is valid for LO mode. Otherwise if $E_{\tau,k} < G_{\tau,k}(\mathsf{LO})$, the message k is no longer valid in LO mode (i.e., it shows HI mode behavior). In addition to this inequality [4], if $C_{\tau,k} \leq C_\tau(\mathsf{LO})$, where $C_\tau(\mathsf{LO})$ is a LO mode message size requirement of flow τ, the message k's behavior is valid for LO mode. Otherwise if $C_{\tau,k} > C_\tau(\mathsf{LO})$, the message k shows HI mode behavior. Once the monitor observes HI mode behavior, it initiates a mode change by utilizing other MC-SDN components presented in the following sections.

B. Mode Change Arrangement

MC-SDN uses switch-driven mode change to effectively reduce and bound the delay of the mode change arrangement. It completely eliminates OpenFlow communication between the controller and switches, which is the root cause of the delay. Instead, it employs the minimal communication between switches, which imposes only a little delay.

When a switch detects a mode violation, its mode change arranger requests all other switches to change to the new mode by sending the highest-priority signal packets to all ports (i.e., signal flooding). Once the signal reaches another switch, the switch becomes aware of the mode change and propagates it again. Such signal propagation takes only a short delay to transmit small packets and even can be hidden by overlapping with the transmission of the mode-violating packets.

In the controller-driven mode change paradigm, the key role of the SDN controller is to distribute the packet forwarding rules of a new mode to all switches, which incurs a significant delay in the mode change arrangement. For instance, it can take up to 50 ms in our preliminary experiment shown in Section III-A. In order to exclude such a delay completely in a mode change, MC-SDN caches new mode rules in advance. When the system starts or a new flow joins the system, the controller deploys not only LO mode rules but also HI mode rules to each switch. As a default, each switch then equips its forwarding table with LO mode rules and stores HI mode rules into a separate place, called *shadow table*. When the switch changes to HI mode, it no longer downloads HI mode rules from the remote controller since it can use the HI rules cached in the shadow table.

This way, it effectively reduces the delay of this step to the signal propagation latency, which is much shorter, since it completely eliminates communication with the controller. As shown in Fig. 1(c), switches are able to immediately proceed to the next step as soon as propagating signal packets. Furthermore, it makes the delay in this step much more predictable, since it goes through only data planes, which is much simpler than the complicated software layers of the SDN controller.

C. New Rule Update

The idea of storing HI mode rules proactively in a shadow table allows to eliminate delays in external OpenFlow communication with the controller. Yet, it can incur a significant delay to update a forwarding table with the HI rules stored in the shadow table due to the internal structure of a switch. The switch often has a multilayered architecture to effectively support multiple protocols and standards. It typically places SDN protocol processing (i.e., OpenFlow processing) on one layer and packet forwarding on another (with a forwarding table). If the shadow table is placed on a different layer from the one where packet forwarding is actually performed, it needs to go through cross-layer internal communication within a switch, which causes non-negligible delays. Furthermore, such delays increase when the forwarding table is updated on demand. Thus, MC-SDN places the shadow table into where packet forwarding is actually performed (e.g., the *datapath* of OVS), and updates the forwarding table without any cross-layer communication. With this design principle, the delay of the rule update step is reduced to the data copy cost between two tables, which is much faster and easy to bound.

D. Out-of-Mode Packet Handling

MC-SDN proposes an advanced queueing feature to reduce the delay in the out-of-mode packet handling step. *MC queue controller* enables to apply HI mode rules to out-of-mode packets, and thereby each switch no longer requires to wait until all out-of-mode packets have been transmitted. To do this, it extends the priority queue; it enables a switch to hook enqueued packets before transmitting them. After all HI rules

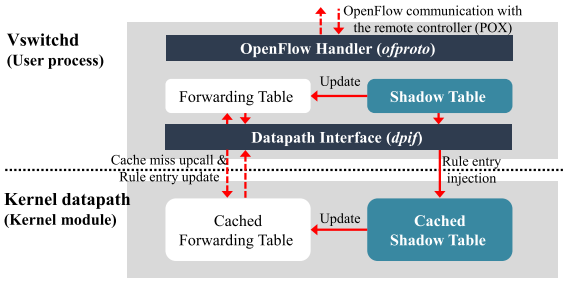


Fig. 4. MC-SDN implementation on OVS.

have been updated, the MC queue controller hooks all packets in the priority queue, and applies new (HI) rules to those packets. Those packets could be *discarded* or *requeued*, according to the HI rules. Note that a flow has a differentiated policy according to the mode; a flow could be forwarded in the LO mode but dropped (or assigned a lower priority) in HI mode. The MC queue controller allows to handle out-of-mode packets with a much shorter delay. In particular, it would be much effective for the system which uses a low bandwidth network link.

V. Implementation

This section discusses implementation issues for MC-SDN, in particular, for OVS [18], [25], which is the de facto standard software switch for OpenFlow implementation.

Target System: We have implemented MC-SDN on top of OVS version 2.4.90, the POX network controller [26], and Linux version 3.10.107. It is worthwhile to describe the internal structure of OVS for ease of understanding this section. As shown in Fig. 4, OVS consists of two components, *vswitchd* and *kernel datapath*. The *vswitchd* is a user process in charge of switch management, and it contains a *forwarding table*, OpenFlow Handler (*ofproto*) for communication with the remote controller, and datapath interface (*dpif*) for communication with the kernel datapath. The *kernel datapath* is a Linux kernel module responsible for packet forwarding. In order to maximize packet forwarding throughput, it has a *cached forwarding table* which holds a subset of rules in the *forwarding table*. Those OVS internals closely interact with each other.

A. Shadow Table for New Rule Update

The shadow table is a key component in reducing mode change delays, and it is one of the most challenging parts to implement due to the complicated structure of forwarding tables. Upon receiving an incoming packet, the kernel datapath first looks up its cached forwarding table; if it cannot find a matching rule, it notifies a cache-miss (i.e., `MISS_UPCALL`) to *vswitchd* and brings the matching rule from it (see Fig. 4). According to the design principle of the shadow table, it should be located in the kernel datapath and be able to directly update the cached forwarding table. Therefore, as shown in Fig. 4, we have implemented the *shadow table* and the *cached shadow table* on the vswitchd and the kernel datapath, respectively.

Then, it raises an issue of how to update the cached shadow table in the kernel datapath. A naive approach is to simply invalidate the rules in the cached shadow table and to update them upon cache misses. However, this introduces non-negligible delays, which cannot overlap with other delay factors. Thus, we updated the cached shadow table proactively, by directly injecting (or removing) rules into the cached shadow table with no cache miss/update protocol. To this end, we extended *dpif*, Linux *netlink* interfaces, and datapath callback functions. Since the vanilla datapath has a unique cached forwarding table, all *dpif* have been designed for a single table. We extended all *dpif* to have a designator to distinguish which table (e.g., cached forwarding or cached shadow tables) to be accessed. In addition, since *dpif* communicate with the kernel datapath through Linux *netlink* interfaces, we also extended *netlink* interfaces with new messages containing the table designator field. And, when the kernel datapath receives a netlink message, it executes a callback function that corresponds to the message. We also extended the callback functions to process the table designator; the extended functions can access (e.g., add/modify/remove entries) to the cached shadow table.

Furthermore, we also implemented new OpenFlow commands, `OFPFC_ADD_SHADOW`, `OFPFC_MOD_SHADOW`, and `OFPFC_DEL_SHADOW`, which support to add, modify, and remove a rule entry in the shadow table, respectively. With those extensions, the cached shadow table can hold HI mode rules within the kernel datapath in advance of a mode change.

B. OpenFlow Extensions for Mode Change Arrangement

We implemented the features of flow behavior monitoring and mode change arrangement as new functions in the kernel datapath with well-defined interfaces for high performance and ease of management. To this end, we implemented them as OpenFlow *actions* through OpenFlow experimenter extension. This demands a great deal of effort since it requires to extend all SDN layers, including new POX APIs, extended message encoders/decoders/handlers in *ofproto* and *dpif*, and the logic for the new actions themselves. To this end, we: 1) defined the new commands with proper headers and fields; 2) implemented encoders and decoders for the new commands in *ofproto*; 3) extended *dpif* to deliver the new commands to the kernel datapath; and 4) implemented the logic for actions themselves into the kernel datapath. In addition, we also extended the OpenFlow handler logic in POX to define new commands and implement encoders/decoders for them. To achieve this, our implementation added around 4000 and 2000 lines of code into OVS and POX, respectively.

Flow behavior monitoring was implemented as an OpenFlow action, named `OFPAT_FLOW_MONITOR`. This action can be combined with a forwarding action such as `OFPAT_ENQUEUE`; thereby, it can check packets before forwarding them. When the first packet of each message arrives, the action executes the monitoring logic presented in Algorithm 1. Note that the monitor action is running in the kernel interrupt context, it can fully utilize Linux socket buffer structures and libraries; it can get a packet arrival time through

the socket buffer time stamp, and directly access to the shim-header through the socket buffer. Executing the monitor would impose a delay, but this delay is small enough to be hidden between packet transmissions (see Section VIII-D for more details).

Mode change arrangement was implemented as another OpenFlow action, named `OFPAT_MODE_ARRANGE`. Upon detecting a mode violation (or receiving a mode change signal), the arranger starts a mode change. The arranger propagates a mode change signal packet containing a predefined L2 header (i.e., 0x0F00 in `EtherType` field); each switch distinguishes the signal with that field. It also includes *switch-id* field to avoid broadcast storms by examining redundant signals. The arranger is also running in the kernel interrupt context; it can generate a signal packet through new socket buffer allocation API. After the signal propagation, the arranger traverses the cached shadow table and copies each entry into the cached forwarding table.

C. MC Queue Controller for Out-of-Mode Packet Handling

During a mode change, MC-SDN rearranges queues to drop out of mode packets or demote their priorities. MC-SDN utilizes the PRIO Linux queuing discipline (*TC-PRIO*) [21], which has a number of child queues and ensures to transmit packets only if all higher priority child queues have been empty. Extending the queueing discipline is not trivial, since small changes can yield serious side effects such as throughput degradation. We placed a hooking routine between dequeueing and transmitting packets; after the rule update is done, MC-SDN dequeues and hooks packets in the queue, and applies new rules to those hooked packets. They could be *requeued* or *discarded* depending on the new rules. Since each child queue in the PRIO queueing discipline is nothing but a simple first-input, first-output (FIFO) queue, dequeueing packets only imposes a very small overhead. In addition, it also incurs very little overhead to requeue and discard packets, since they are implemented as lightweight pointer copy and memory free operations, respectively (see Section VI-B for overhead details).

To avoid side effects on normal packet forwarding performance, we never modified the queueing logic itself, and carefully maintained the consistency of internal data such as packet counters. Section VIII-D presents experimental results that indicate the MC queue controller imposes a negligible effect on the network performance (i.e., normal packet forwarding throughput) even together with the flow behavior monitor.

VI. Mode Change Delay Analysis

In this section, we derive an upper bound of the mode change delay of MC-SDN.

A. Analytic Bound

As we explained in Section III-B, the mode change delay of MC-SDN consists of three components, and therefore we can express the worst-case mode change delay of MC-SDN (denoted by D_{mc}) as follows:

$$D_{mc} = D_{\text{arrange}} + D_{\text{update}} + D_{q-\text{handle}} \tag{1}$$

where D_{arrange}, D_{update}, and $D_{q-\text{handle}}$ denote the worst-case delays of mode change arrangement, new rule update, and out-of-mode packet handling, respectively. We now investigate individual delay components.

Since mode change arrangement delay is the time to propagate signal packets to all switches in the network, its upper bound D_{arrange} can be calculated by the worst-case delay on each hop, multiplied by the maximum hop distance to propagate the mode change signal (denoted by N_{link}). Considering the worst-case delay on each hop can be expressed as the sum of the worst-case delay of transmission, propagation, queueing, processing, and packet flooding overhead (denoted by d_{trans}, d_{prop}, d_{queue}, d_{proc}, and d_{flood}, respectively), D_{arrange} can be computed as follows:

$$D_{\text{arrange}} = \left(d_{\text{trans}} + d_{\text{prop}} + d_{\text{queue}} + d_{\text{proc}} + d_{\text{flood}}\right) \cdot N_{\text{link}}. \tag{2}$$

Here, d_{trans}, d_{prop}, and d_{queue} are determined by physical properties of the network system such as link bandwidth, physical link length, link propagation speed, and the number of non-preemptible packets. Other delay components d_{proc} and d_{flood} are dependent on switch architecture.

With the MC-SDN design principles, the new rule update and the out-of-mode packet handling steps become nothing but iterations of simple operations; D_{update} and $D_{q-\text{handle}}$ can be expressed as a function of the number of rules to update (N_{rule}) and out-of-mode packets (N_{packet}), respectively,

$$D_{\text{update}} = d_{\text{copy}} \cdot N_{\text{rule}} + d_{u-\text{misc}} \tag{3}$$

$$D_{q-\text{handle}} = d_{q-\text{handle}} \cdot N_{\text{packet}} + d_{q-\text{misc}} \tag{4}$$

where d_{copy} is the maximum required time to copy each rule from the shadow table to the forwarding table; $d_{u-\text{misc}}$ is an additional rule-update overhead regardless of N_{rule}; $d_{q-\text{handle}}$ is the maximum required time to handle each out-of-mode packet; and $d_{q-\text{misc}}$ is an additional overhead regardless of N_{packet}. The additional overheads $d_{u-\text{misc}}$ and $d_{q-\text{misc}}$ include execution costs to initialize and finalize the iterations, for example, referring internal data structures to get the accesses of the tables and the queue.

We calculate the worst-case mode change delay of MC-SDN (denoted by D_{mc}) by decomposing it into the three components, each of which also consists of several computable subcomponents. Once we calculate an upper bound on each component, we can derive an upper bound on D_{mc} by summing each term. We can then incorporate D_{mc} into target schedulability tests, by adding D_{mc} to the transmission time of HI flows.

B. Upper Bound of Delay Components

In this section, we detail how to calculate each delay component in the delay bound, using the real network testbed. The testbed consists of several *Odroid-XU4* [27] boards equipped with Realtek r8152 [28] USB Ethernet interfaces (see Section VIII for details). Note that some delay bounds (e.g.,

TABLE II
UPPER BOUNDS OF MODE CHANGE ARRANGEMENT DELAY COMPONENTS

Component	Bound (μs)	Average (μs)	Stdev (μs)	Number of samples
Overall	1453.45	-	-	-
d_{prop}	0.53	-	-	-
d_{trans}	4.6	-	-	-
d_{queue}	600.64	-	-	-
d_{proc}	766.90	764	41	1300
d_{flood}	80.78	80.02	1.61	30

TABLE III
UPPER BOUNDS OF NEW RULE UPDATE AND OUT-OF-MODE PACKET HANDLING DELAY COMPONENTS

Component	Bound (μs)	Average (μs)	Stdev (μs)	Number of samples
d_{copy}	3.95	3.80	0.94	280
d_{u-misc}	50.04	49.72	0.68	30
$d_{q-handle}$	1.06	1.04	0.44	5200
d_{q-misc}	2.07	1.92	0.32	30

d_{trans}, d_{prop}, and d_{queue}) are analytically derived based on the physical properties (e.g., link bandwidth, physical link length, and link propagation speed) shown in the testbed. Other delay bounds are empirically derived at the 99.5% confidence level based on execution samples in the testbed.

1) Mode Change Arrangement: Mode change arrangement delay D_{arrange} consists of five components as presented in Table II, which can be modeled as a typical end-to-end network delay. The link propagation delay d_{prop} is known as 530 ns at 100 m of Cat.5e UTP Ethernet Cable [29]. We use this value as a safe upper bound, since our system only uses an up to two meters long Ethernet cable. The link transmission delay d_{trans} is calculated as (packet length/allocated bandwidth). Since a signal packet has the fixed length of 58 bytes and utilizes full network bandwidth of 100 Mb/s (note that it has the highest priority), d_{trans} can be calculated as $(58 * 8 \text{ bits}/100 \text{ Mb/s}) = 4.6\ \mu$s. The queueing delay d_{queue} can be upper bounded by the transmission time of nonpreemptible packets. Once some packets are sent out from the priority queue (by Linux *TC-PRIO* queueing discipline), they are delivered to the device driver buffer and finally transmitted to the NIC hardware in an FIFO order. Although the mode change signal packet has the highest priority, it may be blocked by the packets already placed in either the device driver or the NIC hardware (i.e., r8152 [28]) ahead of the signal packet. Considering that the device driver and the NIC hardware can store packets up to 5460 and 2048 bytes, respectively [30], d_{queue} can be calculated as $[((5460 + 2048) * 8 \text{ bits})/(100 \text{ Mb/s})] = 600.64\ \mu$s. The processing delay d_{proc} and the packet flooding overhead d_{flood} are estimated as the maximum values in the 99.5% confidence intervals based on empirically obtained samples. In addition, N_{link} can be upper bounded by the maximum value among the hop distances between any two switch nodes.

2) New Rule Update and Out-of-Mode Packet Handling: The delay components of new rule update and out-of-mode packet handling delays are also estimated as the maximum values in the 99.5% confidence intervals. In order to obtain the execution samples, we measured the time at which each execution starts and finishes within the *datapath* module by using the `getnstimeofday()` kernel function. Table III describes the statistics of the measured samples and the corresponding delay bounds. Note that in out-of-mode packet handling, the *discard* operation takes much longer time than the *requeue* operation does, since the former requires to free some memory space while the later only needs a simple pointer copy operation. Therefore, $d_{q-\text{handle}}$ is upper bounded by the discard operation instead of the requeue operation. In addition, N_{rule} and N_{packet} can be upper bounded by the maximum number of rules and the maximum queue length in each switch, respectively.

The proposed upper bound has the 99.5% confidence level under the assumption that the population has a normal distribution. For a higher assurance, we can apply static WCET analysis techniques [31].

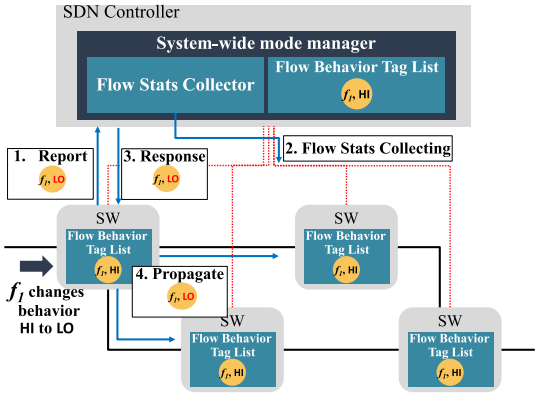


Fig. 5. System-wide mode management protocol supporting HI to LO mode change.

VII. MODE MANAGEMENT PROTOCOL DESIGN

While Section IV provides the core system design necessary for mode changes, we need a detailed protocol design how mode changes operate using the system, in particular, when the mode is changed not only LO to HI but also HI to LO. The former (i.e., LO to HI mode change) is already presented in Section IV; that is, a switch immediately triggers the mode change upon any mode violation detection. However, the latter (i.e., HI to LO mode change) raises several research questions, including: 1) when to change the system mode from HI to LO and 2) how to manage the system mode to preserve the system consistency. In this section, we propose *system-wide mode manager* and the mode management protocols to address those issues.

A. Challenges to Return to LO Mode

The system can return to the LO mode, if all flows no longer show HI behavior. The first challenge is a *safe* HI to LO mode change, to guarantee real-time requirements of HI flows despite the mode change to LO. Recall the system mode definition shown in Section II as follows. The system should change the mode to HI, when *at least one* flow shows HI behavior; thereby, a switch should immediately trigger a mode change to HI once it sees any HI behavior. In contrast,

Algorithm 2 Flow Behavior Monitor With Tag

1: **function** `behavior`$(E, G, C, C(\mathsf{LO}))$
2: **if** $E < G$ **or** $C > C(\mathsf{LO})$ **then**
3: **return** HI
4: **else**
5: **return** LO
6: **end if**
7: **end function**
When the message k of the flow τ arrives at $E_{\tau,k}$,
8: $G_{\tau,k}(\mathsf{LO}) \leftarrow max\{G_{\tau,k-1}(\mathsf{LO}), E_{\tau,k-1} - J_\tau\} + T_\tau(\mathsf{LO})$
9: **if** $\sigma_\tau \notin \Phi$ **and**
`behavior`$(E_{\tau,k}, G_{\tau,k}(\mathsf{LO}), C_{\tau,k}, C_\tau(\mathsf{LO})) = \mathsf{HI}$ **then**
10: propagate an `add_tag`(σ_τ)
11: **else if** $\sigma_\tau \in \Phi$ **and**
`behavior`$(E_{\tau,k}, G_{\tau,k}(\mathsf{LO}), C_{\tau,k}, C_\tau(\mathsf{LO})) = \mathsf{LO}$ **then**
12: report LO behavior to the controller with σ_τ
13: upon receiving the response, propagate an `pop_tag`(σ_τ)
14: **end if**

Algorithm 3 Mode Change Arranger With Tag

Upon receiving the signal with a tag σ_τ,
1: **if** the signal is `add_tag` **then**
2: $\Phi \leftarrow \Phi \cup \{\sigma_\tau\}$
3: **if** $|\Phi| > 0$ **and** current mode is LO **then**
4: mode change to HI
5: **end if**
6: **else if** the signal is `pop_tag` **then**
7: $\Phi \leftarrow \Phi \setminus \{\sigma_\tau\}$
8: **if** $\Phi = \emptyset$ **and** current mode is HI **then**
9: mode change to LO
10: **end if**
11: **end if**

the system can return to LO mode only if *all* flows show LO behavior; the system must make sure that no messages show HI behavior before triggering a mode change to LO (i.e., *safe* mode change), which is the first challenge.

The second challenge is that the system should preserve the system-wide mode consistency against conflicting mode change signals. Let us assume that two distinct flows simultaneously change their behavior to LO and HI, respectively. The system should be in HI mode by the system mode definition. However, each switch receives two distinct mode change signals (to HI and LO, respectively); it then determines the mode by the signal received later. The arrival times of two signals could be different on each switch because of the signal propagation delay. As a result, the mode of each switch could be different with each other; this inconsistent system mode may result in a severe penalty on highly critical flows.

B. Safe HI to LO Mode Change Protocol

MC-SDN supports a safe HI to LO mode change protocol by using an additional component, named *flow stats collector*, which gathers flow statistics and helps to ensure that all flows show LO behavior. As shown in Fig. 5, the HI to LO mode change protocol is as follows.

1) *Report,* when the flow f_1 changes its behavior from HI to LO at the ith message M_i, the flow behavior monitor reports it to the controller.
2) *Flow Stats Collecting,* upon receiving the report, the flow stats collector gathers statistics of the flow (i.e., packet/byte counters) from the switches along its route. With the statistics, the controller can check whether all messages until M_{i-1} have already gone outside of the network.
3) *Response,* if the flow stats collector confirms that f_1 is valid in LO mode (i.e., no more HI messages in the network), the controller responses it to the switch.
4) *Propagate,* once receiving the response, the mode change arranger of the switch starts to propagate the HI to LO mode change signal.

This protocol takes a short delay and defers triggering the HI to LO mode change. Due to the protocol delay, flows could be scheduled by HI mode rules although they show LO behavior. However, despite this delay, timing requirements of HI flows are always guaranteed (note that LO behavior utilizes less resources than HI behavior).

C. Preserving Mode Consistency

For system-wide mode consistency, each switch should maintain the context of flow behavior changes; MC-SDN introduces *flow behavior tag*, which contains the behavior context of each flow, and a mode management protocol exploiting the tag. The flow behavior tag consists of the changed behavior; it also contains the flow and message IDs that show the behavior change. Algorithms 2 and 3 present how the flow behavior monitor and the mode change arranger work with the tag, respectively.

As shown in Algorithm 2, when a switch detects a behavior change of a flow τ, it then generates a flow behavior tag σ_τ and propagates the signal with the tag to other switches. Switches and the controller maintain the *flow behavior tag list* Φ that keeps the tag for flows showing HI behavior. The monitor can determine changed behavior according to the tag list and flow behavior. When it sees HI behavior of a flow τ without a tag σ_τ in the list, it then propagates an `add_tag` signal to all switches (lines 9 and 10 in Algorithm 2). In contrast, when it sees LO behavior of a flow τ with a tag σ_τ in the list, it follows the safe HI to LO mode change protocol and propagates a `pop_tag` signal (lines 11–13 in Algorithm 2).

Algorithm 3 presents the mode change arranger; it changes the mode according to the received signal and the tag list. When the arranger receives an `add_tag` signal, it first adds the tag σ_τ into the list; if the mode is LO and the list has at least one tag, it then changes (or stay) into HI mode (lines 1–4 in Algorithm 3). On the other hand, when the arranger receives a `pop_tag` signal, it first removes the tag σ_τ from the list; it then checks whether the list is empty. An empty tag list implies that all flows show LO behavior. Thereby, if the mode is HI and the tag list is empty, it changes the mode to LO (lines 6–9 in Algorithm 3).

This protocol allows to preserve system-wide mode consistency by eliminating any confusion from concurrent flow behavior changes. Let us assume that a system tries to return to LO mode upon the behavior change of the flow f_1. Right after propagating the LO mode change signal, suppose another flow f_2 shows HI behavior. In this case, the system should stay in

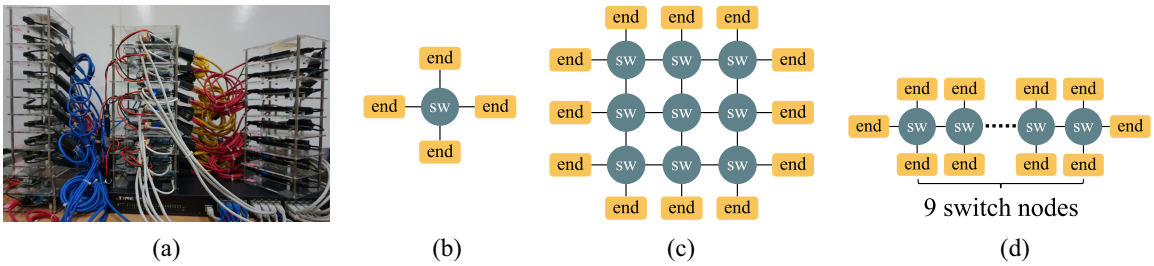


Fig. 6. Network testbed and various topology. (a) Testbed photograph. (b) Star topology. (c) Grid topology. (d) Linear topology.

HI mode. However, the LO mode change request could arrive at a switch after the decision (i.e., stay in HI mode) has taken place, due to arbitrary network delays. If the switch does not apply the tag list, it has no way to distinguish whether the LO mode change signal is valid or not. Then, it changes the mode to LO, and f_2 may miss its deadline. The tag list eliminates this situation. Since HI and LO mode changes are conducted according to the existing tags in the list, the system ensures that the mode is HI if any flow shows HI behavior. In the above case, the system will stay in HI mode regardless of the arrival time of LO mode change signal made by f_1, because the tag list has the context which indicates that f_2 shows HI behavior.

D. OpenFlow Extensions for Mode Management Protocol

We implemented the system-wide mode manager as an OpenFlow extension, and realized the proposed protocol. The OVS and POX are extended with new OpenFlow commands `OFPT_MC_REPORT` and `OFPT_MC_RESPONSE` that carry the flow behavior tag for the HI to LO mode change. The flow behavior tag list is simply implemented as an internal data structure of OVS and POX. In addition, the flow stats collector is implemented as a POX module, which periodically gathers designated flow statistics through the `STATS_REQUEST` OpenFlow command.

VIII. Evaluation

In this section, we evaluate MC-SDN by answering the following questions.

1) How much delay does MC-SDN incur during mode change? (Section VIII-A).
2) How does the mode change affect end-to-end transmission time under MC-SDN? (Section VIII-B).
3) How does the mode management protocol improve resource efficiency? (Section VIII-C).
4) How much overhead does MC-SDN impose for packet forwarding? (Section VIII-D).

Experimental Setup: Experiments were performed on a network testbed [see Fig. 6(a)], which consists of 20 end nodes (*Beaglebone-Black* [32] boards), nine software switches (*Odroid-XU4*, [27] boards), and an SDN controller (a desktop with Intel i5-3750 and 32-GB RAM). To increase the connectivity of switch nodes, we equipped each switch node with additional four USB Ethernet interfaces (Realtek r8152 [28]) with a USB2.0 hub (Belkin F4U040kr). Switch nodes and end nodes were connected via 100 Mb/s Ethernet, and each switch had a dedicated Ethernet interface for the remote SDN controller.

Metric: We measured *mode change delay* as an elapsed time from the instant at which a switch detects mode violation to the instant at which all switches finish their forwarding tables with new mode rules and penalize out-of-mode packets. We also measured *end-to-end transmission time* as the time taken to transmit a message from its source to a destination. For measurement, all switches and end nodes were synchronized by network time protocol (NTP) with an accuracy of less than 1 ms.

Mode-Based Scheduling: Unless stated otherwise, each switch prioritizes packets according to RM while scheduling both HI and LO flows in LO mode but dropping LO flows in HI mode. Note that in LO mode, LO flows could be assigned higher-priorities than HI flows depending on their periods. For comparison, Std-SDN indicates the controller-driven mode change approach based on the standard SDN protocol, as described in Section III-A. MC-Agnostic indicates a non-MC approach that does not conduct mode change; it keeps using RM scheduling without dropping any LO flows even though a HI flow shows HI behavior.

Network Topology: Experiments were performed on various network topologies: *star*, *grid*, and *linear* as shown in Fig. 6(b)–(d), respectively,

A. Mode Change Delay

We ran various experiments to examine how well MC-SDN addresses several delay factors of a mode change. During the experiments, we ran a single HI flow that transmits a message of up to 180 kB in LO mode with a period of 100 ms. After a few seconds of each experiment, the HI flow doubled its message size, and the system conducts a mode change to HI mode.

Fig. 7(a) and (b) shows the mode change delays of Std-SDN and MC-SDN over different network topologies, while the HI flow went through all switches in each network topology. In the figures, the labels of 1, 50, and 100 on the x-axis indicate how many new rules to update in the forwarding table, respectively; note that 100+ on the x-axis implies 100 rules to update *with* additional out-of-mode packets, to be detailed in the last paragraph of this section. The figures show 20 measurements while each gray box covers the 25th–75th percentiles with the line inside indicating the 50th percentile, and the error bar represents the minimum and maximum delays. In every scenario, the figures show that Std-SDN incurs significantly larger and

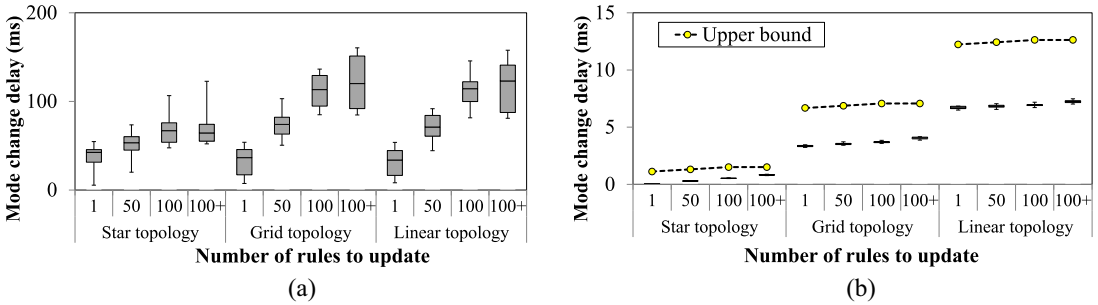


Fig. 7. Mode change delays with varying the topology, the number of rules to update, and the presence of out-of-mode packets. (a) Std-SDN. (b) MC-SDN.

highly fluctuating mode change delays than MC-SDN does (note the different scales on the *y*-axis).

Fig. 7(b) also depicts the delay bound as a dotted-line; we calculated the bound with the values presented in Section VI-B. To calculate the bound, we used N_{link} of 0, 4, and 8 according to the topology, N_{packet} of 1000 based on the maximum queue length of network interfaces, and N_{rule} identical to the number of flows to update. The figure shows that MC-SDN not only effectively reduces the mode change delay, but also strictly limits the delay to the upper bound.

1) Mode Change Arrangement: Fig. 7 shows that the distributed way of mode change arrangement of MC-SDN yields much shorter delays than the centralized way of Std-SDN. In particular, even though it needs to update the minimal number of rules (i.e., only one rule), the figures show that MC-SDN effectively reduces delays by an order of magnitude in the mode change arrangement step while eliminating OpenFlow communication. We note that the delay of MC-SDN includes delays in mode change propagation along with switches. In the grid and linear topologies, it should propagate up to 4 and 8 hops, and the delay increases as its propagation distance grows.[3]

2) New Rule Update: Fig. 7 shows that the delay generally increases when each switch has a larger number of rules to update, but in a different order of magnitude between Std-SDN and MC-SDN. When updating 50 and 100 rules in the mode change, Std-SDN imposes additional long delays (up to 92 ms) that vary significantly, while MC-SDN adds only 0.2–0.3 ms. This is because MC-SDN updates the forwarding table with the information stored in the shadow table by eliminating external communication with the remote SDN controller and minimizing intraswitch cross-layer communication.

3) Out-of-Mode Packet Handling: In order to evaluate the out-of-mode packet handling, we ran experiments with two additional LO flows that share the links with the HI flow, where 100 rules are updated in mode change. Each LO flow has the period of 100 ms and the size of 490 kB. Fig. 7 shows the results on the *x*-axis labeled 100+; note that 100+ represents the case with 100 rules to update *and* additional out-of-mode packets generated by the additional LO flows. It shows that while Std-SDN adds high fluctuations in the order of tens of millisecond (up to 24 ms), MC-SDN increases the delay only in microseconds (up to 310 μs) by dropping LO flows out of the queue.

[3]We note that the propagation delay is slightly high (i.e., about 0.7 ms per hop) due to the poor performance of USB Ethernet in our setup; it could be significantly lowered when just using PCI or on-board Ethernet cards.

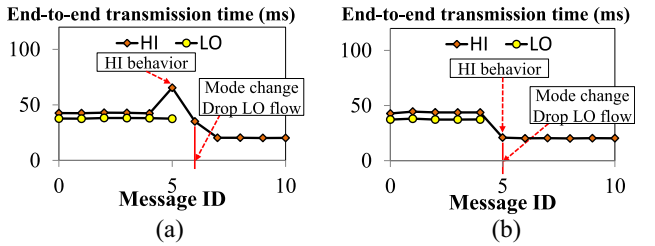


Fig. 8. End-to-end transmission time of each message with the mode change. (a) Std-SDN. (b) MC-SDN.

TABLE IV
SPECIFICATIONS OF THE TWO MC FLOWS

	L_i	T_i (ms)	C_i(LO) ($KBytes$)	C_i(HI) ($KBytes$)	Priority LO	Priority HI
f_1	HI	50	120	240	low	low
f_2	LO	50	450	450	high	drop

B. End-to-End Transmission Time

In this section, we evaluate the effect of mode change delay on end-to-end transmission time over various experimental scenarios.

1) Two Contending Flows: This experiment illustrates how the mode change delay affects the end-to-end transmission times of HI and LO flows that are contending with each other. We generated two flows as described in Table IV, on the linear topology network shown in Fig. 6(d); their routes were overlapped so that the LO flow interfered the HI flow. The *deadline* of each flow was identical to the *period* of that. Fig. 8 depicts the end-to-end transmission time of each message of two flows with a mode change. Note that the HI flow doubled the message size from the message of ID 5 until the last one.

Fig. 8(a) shows that the HI flow, especially the message of ID 5, suffers from an unintended increase of the end-to-end transmission time. The mode change delay of Std-SDN causes this increase; although the switch can be aware of the LO mode violation from the message of ID 5, it takes a while to complete the mode change. During the mode change delay, the LO flow's message of ID 5, which must be dropped, interferes the HI flow. This interference incurs a deadline miss of the HI flow. Besides, since the interference caused by the

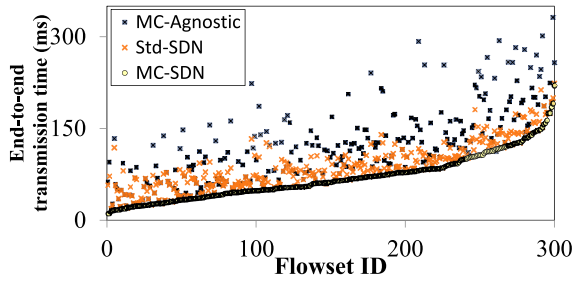


Fig. 9. End-to-end transmission times of the messages that show HI behavior at the first time.

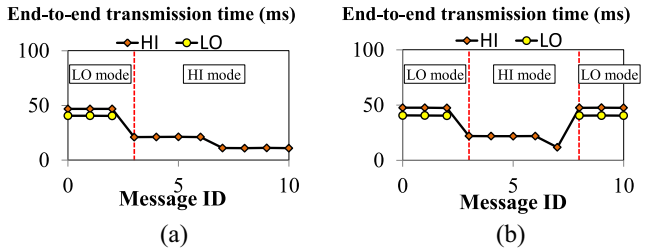


Fig. 10. End-to-end transmission time of each message with or without return to LO mode. (a) Without the return to LO mode; all LO messages after ID 3 are dropped. (b) With the return to LO mode; LO messages (IDs 3–7) are dropped.

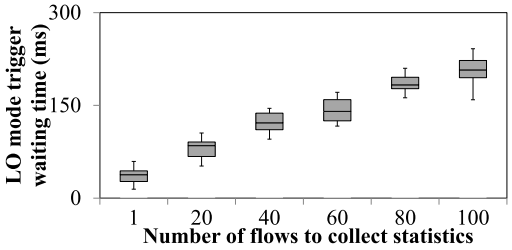


Fig. 11. Delay of the flow stats collecting step with varying the number of flows to collect statistics.

mode change delay is unpredictable, it is impossible to guarantee the end-to-end transmission times of HI flows which are critical to the system safety. In contrast, Fig. 8(b) shows that MC-SDN properly handles the HI flow with subtle, almost zero interference during the mode change. Thanks to the MC-SDN design, the switch timely completes the mode change and drops the LO flow's messages including ID 5; thus the HI flow does not suffer from an unintended interference. In addition, since MC-SDN not only reduces the delay but also strictly bounds it, the end-to-end transmission time could be predictable even if a mode change happens. Consequently, this predictability can help to guarantee real-time requirements of MC flows.

2) Synthetic Flow Set: To show the effectiveness of MC-SDN for general cases, we conducted the experiment with randomly synthesized flow set; on each experiment, we generated a set of up to 16 real-time flows on the grid topology [shown in Fig. 6(c)] as follows. For each flow, we selected its period randomly between 10 and 200 ms, a message *size* that can be transmitted randomly between 10% and 40% of the period on a 100-Mb/s link, and its *criticality* to be HI with the probability of 33%. In addition, we randomly determined its source and destination nodes, and a route was determined as a shortest path inbetween. During each experiment, one HI flow was assigned to trigger a mode change by sending a message M twice as big, and Fig. 9 plots the end-to-end transmission times of the message M on MC-SDN, Std-SDN, and MC-Agnostic, respectively, in increasing order of MC-SDN's measurements for the ease of presentation.

As shown in Fig. 9, MC-SDN always results in end-to-end transmission times that are smaller than or equal to the ones of MC-Agnostic and Std-SDN. On average, MC-Agnostic and Std-SDN incur 46 and 20 ms longer end-to-end transmission times than MC-SDN, respectively; in the worst case, they show 211 and 102 ms longer results than MC-SDN, respectively. The main reason of the longer transmission time is the unintended interference by LO flows which should be dropped in the HI mode. Although Std-SDN supports a mode change, it is impossible to bound the interference due to the unpredictability of the mode change delay; note that the transmission time difference between Std-SDN and MC-SDN widely varies in Fig. 9. In some cases (34 out of 300 cases), we observe that all systems result in identical transmission times; this is because there are no LO flows which have higher priority than the message M. Consequently, Fig. 9 implies that MC-SDN effectively reduces the mode change delay and then improves schedulability of MC flows, in general cases.

C. Mode Management Protocol

This section shows the effectiveness of the MC-SDN mode management protocol with the following aspects: how the HI to LO mode change enhances the system resource efficiency, how long the LO mode change takes, and how well MC-SDN preserves the system-wide mode consistency.

1) Benefit of the Return to LO Mode: To show the benefit of HI to LO mode change, we ran an experiment with two flows that have identical specifications used in Section VIII-B (refer to Table IV). In this experiment, the HI flow doubled the size of messages of IDs 3–6; thereby the system could return to LO mode after transmitting the HI message of ID 6.

Fig. 10(a) shows the system only supporting LO to HI mode change. Since the system stays in HI mode once it observes HI behavior, all LO messages after ID 3 are dropped even though the HI messages after ID 6 become valid in LO mode. Consequently, the system wastes the available resources. In contrast, Fig. 10(b) shows the system also supporting HI to LO mode change. The system returns to LO mode and transmits the LO flow again, after the HI flow shows LO behavior (from ID 7). As a result, the system can efficiently utilize the resources; it can transmit more LO messages.

It is worth noting that the LO mode change takes a while to complete from the LO behavior detection (see the messages of IDs 7 and 8). This is because the *flow stats collecting* step is required to trigger the LO mode change (see Fig. 5); Fig. 11 shows this delay in detail.

2) Delay of Collecting Flow Stats: We measured the delay in the flow stats collecting step with varying the number of flows to collect statistics. Fig. 11 shows that this step only

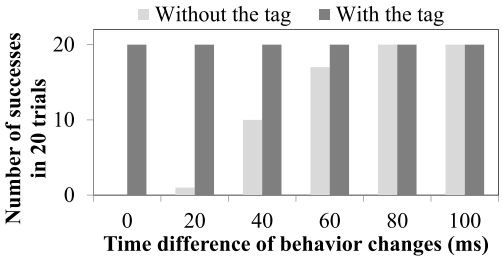


Fig. 12. Number of trials that successfully preserve the system-wide mode consistency.

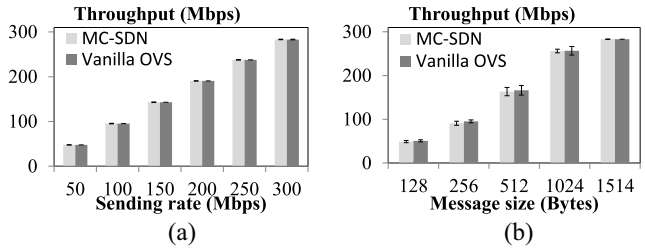


Fig. 13. Packet forwarding throughput of MC-SDN. With varying the (a) sending rate and (b) message size.

takes hundreds of milliseconds even in an extreme case (i.e., 100 flows simultaneously return to LO mode). Note that the figure represents 20 trials, where each gray box covers the 25th–75th percentiles with the line inside indicating the 50th percentile, and the error bar represents the minimum and maximum delays. This delay is affordable because it never incurs any interference to HI messages; it only penalizes few LO messages. Once the HI to LO mode change is triggered after this step, the mode change itself only takes as short as the LO to HI mode change. With this step, MC-SDN ensures a safe mode change from HI to LO with an affordable delay.

3) System-Wide Mode Consistency: To show how well the MC-SDN mode management protocol preserves the system-wide mode consistency, we made two conflicting flow behavior changes simultaneously. In each experiment, two HI flows f_1 and f_2 were generated. At the starting time of each experiment, f_1 doubled its message size; thus the system changed to HI mode. After each experiment started, at a time instant t_1, f_1 reduced the message size to be valid in LO mode. In addition, at t_2, f_2 changed its behavior to HI. We examined the system mode, while varying the time difference of $t_2 - t_1$; since f_2 showed HI behavior after t_2, all switches should be HI mode after t_2.

Fig. 12 depicts the number of successful trials which preserve the mode consistency (i.e., all switches are HI mode) out of 20 trials. We compare two MC-SDN systems, with and without *flow behavior tag list*; MC-SDN with the tag list follows the protocol presented in Section VII, and MC-SDN without the tag list changes the mode whenever it receives any mode change signal.

The figure shows that MC-SDN without the tag list cannot preserve the mode consistency, when $t_2 - t_1$ is short. This is because the LO mode change signal could arrive later than the HI mode change signal, but switches without the tag list have no way to determine whether the delayed LO mode change signal is valid or not. On the other hand, MC-SDN with the tag list always shows a consistent system mode. This is because the tag list allows to effectively validate the mode change signal regardless of the arrival times; it keeps the system in HI mode according to the information that f_2 shows HI behavior.

D. Packet Forwarding Overhead

MC-SDN incurs some overhead in packet forwarding, since each switch applies the *flow behavior monitor* as an OpenFlow action to each packet. In this section, we measured the overhead of MC-SDN with the star topology testbed shown in Fig. 6(b). An end node generated messages with varying the sending rate and message size; the MC-SDN switch monitored the flow behavior, and another end node measured the forwarding throughput. In Fig. 13, each bar plot and error bar represent the average and the standard deviation of 20 trials, respectively, and each trial ran for 5 s. We compare the result with *vanilla OVS*, an unmodified version of OVS. Note that we used 1G USB Ethernet interfaces (Realtek r8153 [33]) for more intensive evaluations with heavy traffic (up to 300 Mb/s, which is the physical limit of our testbed).

Fig. 13(a) shows the throughput with varying the sending rate. Each message was generated with the size of 1514 bytes (i.e., MTU including the Ethernet header). The figure shows that throughput of MC-SDN is comparable to that of vanilla OVS regardless of the sending rate. This is because the packet monitoring action only incurs very small overhead (i.e., the average of 760 ns per packet) that can be effectively hidden in between packet transmissions. Fig. 13(b) depicts the throughput with varying the message size. The sending rate was fixed at 300 Mb/s. Similar to Fig. 13(a), MC-SDN results in comparable performance to vanilla OVS regardless of the message size. Note that the switch cannot forward traffic as fast as the sending rate of 300 Mb/s when the message size is small, due to its low packet processing capability. Despite the heavy workload with the small messages, MC-SDN still hides the overhead effectively.

Besides, other components of MC-SDN except the flow behavior monitor (e.g., MC queue controller) do not affect packet forwarding performance, since they are not involved for normal packet processing. Consequently, the overhead for packet forwarding is negligible.

IX. Case Study: Autonomous Vehicle

In order to show how real world systems benefit from MC-SDN, we conducted a case study, supporting the AEB system, on a 1/10 scaled autonomous vehicle.

A. Autonomous Emergency Braking

AEB is a system that brakes the car in emergency situations by predicting the risk of collisions with detecting obstacles through various sensors. For instance, Jaguar F-PACE provides the AEB feature based on stereo cameras [34]. Since AEB

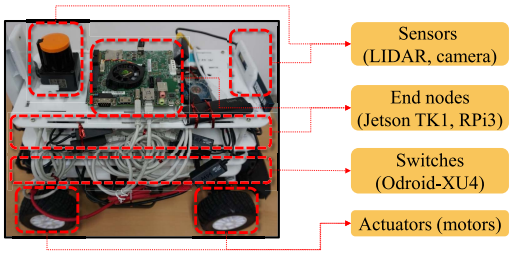


Fig. 14. 1/10 scale autonomous vehicle.

TABLE V
FLOW SET SPECIFICATION IN THE CAR SYSTEM

	L_i	T_i(LO) (ms)	T_i(HI) (ms)	Priority LO	Priority HI	C_i *(KB)*	src.	dst.
LIDAR	HI	25	25	high	high	8	s_1	d_1
STREAM	LO	40	40	mid	drop	432	s_2	d_2
CAM	HI	200	22	low	low	154	s_1	d_1

effectively lowers the risk of car accidents, automakers have agreed to equip it as a standard feature [35].

B. Experimental Setup

We have implemented a 1/10 scaled autonomous car, as shown in Fig. 14, which is extended from the F1/10 autonomous racing platform [36]. The car consisted of the Traxxas Rally 1/10 body with actuators (i.e., motors) [37], sensors including a LIDAR (Hokuyo UST-10LX [38]) and a depth camera (Intel Realsense R200 [39]), Jetson TK1 [40] and Raspberry PI3 [41] boards for end nodes, Odroid-XU4 [27] boards for switch nodes, and an additional Raspberry PI3 for an SDN controller. As shown in Fig. 15(a), each node was connected with each other via 100 Mb/s-Ethernet. And, sensors and actuators were connected to the s_1 and d_1 nodes, respectively, via USB or dedicated interfaces. The car drove itself along the given trajectory, by using the LIDAR-based SLAM [42] and the PID controller for motors. They were implemented as components of the ROSs [19] framework.

As shown in Table V, the car system generated three real-time flows.

1) *LIDAR* for the LIDAR sensor data used for SLAM.
2) *STREAM* for video frames of a user entertainment.
3) *CAM* for image frames from the depth camera used for AEB.

Note that the period of the CAM flow had multiple requirements according to the mode. When the car sees obstacles while moving at high speed, AEB requires a high sensing rate for responsive braking. In contrast, when the car moves on the clear road at low speed, a low sensing rate could be acceptable for AEB. To consider this characteristic, the s_1 node skimmed through depth camera images and adjusted the period; the period had a default value of 200 ms (i.e., LO mode behavior), but it decreased to 22 ms (i.e., HI mode behavior) when depth images contain some obstacles in front of the car. In the LO mode, the car system handled all flows according to the assigned priorities as shown in Table V; on the other hand, in the HI mode, the system dropped the STREAM flow (i.e., a LO flow) to prioritize the LIDAR and the CAM flows (i.e., HI flows).

C. Experimental Scenario

Fig. 15(b) illustrates the experimental scenario. The car was placed 7 m ahead of a wall, and it drove itself toward the wall at the top speed of 2.1 m/s. We restricted the camera's field of view to 2 m, to fix the point where the car can detect the wall. We evaluated the braking performance by observing several points as follows: the detecting point where the wall could be detected by the camera, the braking point where AEB sought to brake the car, and the stopping point where the car completely stopped. We define *perception and reaction*, *braking*, and *total stopping* distances, respectively, as depicted in Fig. 15(b) and use them as performance metrics. To show the effectiveness of MC-SDN, we used two baselines: 1) *LO Only* and 2) *HI Only*. They are static systems which cannot change the forwarding rules. In LO Only, all flows always generated messages according to the LO mode requirement, and thereby all of them were handled with priorities of the LO mode as presented in Table V. On the other hand, in HI Only, all flows operated as the HI mode requirement, and thus the STREAM flow was dropped according to the priority policy of the HI mode.

D. Implication

Fig. 15(c) depicts the braking performance of the car with varying the underlying systems. Each box plot and error bar represent the average and the standard deviation of 30 trials, respectively. In addition, the line graph represents the worst result among the 30 trials. The main factor of the performance is the perception and reaction distance; it depends on how quickly the depth images which contain obstacles could be delivered to the control node (i.e., the d_1 node).

The figure shows that LO Only results in a long stopping distance; due to the long period of the CAM flow, the control node (d_1) cannot be quickly aware of the wall. Although LO Only can efficiently utilize the resource (note that it always serves the STREAM flow), it may hurt the safety of the car system. In contrast, HI Only results in a short stopping distance compared to LO Only. It helps to provide better safety of the car, but it may waste the resource; it cannot serve the STREAM flow. On the other hand, MC-SDN effectively supports MC flows which have multiple requirements. The CAM flow changes its period (from 200 to 22 ms) at the detecting point. MC-SDN timely changes the mode and properly prioritizes the CAM flow while dropping the STREAM flow. As a result, MC-SDN shows a similar stopping distance compared to HI Only. In addition, it can accommodate the STREAM flow before the CAM flow shows HI mode behavior.

This case study implies that MC-SDN effectively realizes MC flow scheduling onto the real world system such as the autonomous car. It enables to balance between two conflicting objectives: 1) efficient resource sharing and 2) safety-critical real-time requirements guarantee.

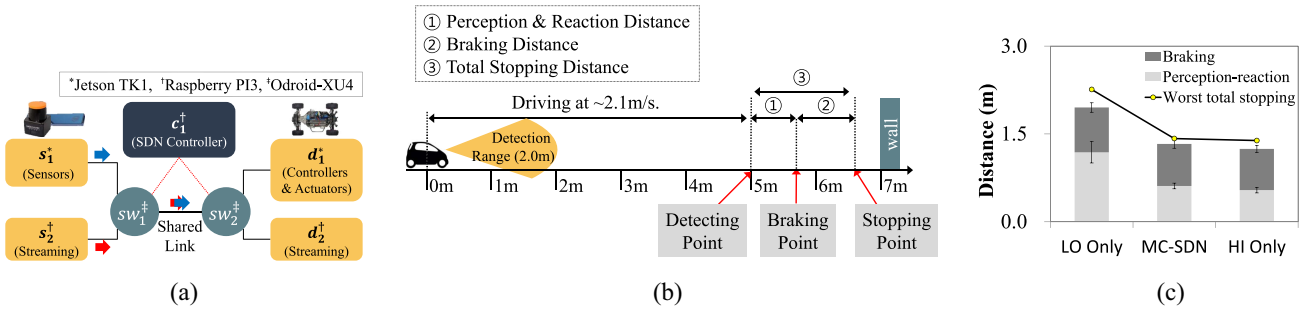


Fig. 15. Evaluation with the 1/10 scale autonomous vehicle. (a) System architecture and topology. (b) Evaluation overview. (c) Stopping distances.

X. New Application With Extension: Fault Resilient Autonomous Vehicle

Beyond the mode-based scheduling for MC systems, MC-SDN is also applicable to other systems which require time-sensitive forwarding rule management. A fault resilient networking system is a suitable application of MC-SDN. The system requires a fault recovery scheme to be done within a short and bounded delay; MC-SDN can support this requirement with a mode-based fault recovery. In this section, we extend MC-SDN and evaluate the control performance against a link fault, in the *autonomous lane keeping* (ALK) scenario.

A. Mode-Based Fault Resilient System

To support a link fault recovery, we made several changes to MC-SDN as follows. We define two system modes, *normal* and *emergency*, according to the link connectivity. The system starts with the normal mode, and it changes into the emergency mode when a link fault occurs. Each switch has backup rules for the emergency mode in its shadow table. In addition, *flow behavior monitor* is replaced by *link fault monitor*, which monitors connectivity of each link and detects a link fault. When it sees any link fault, it immediately triggers a mode change to the emergency mode. It follows the MC-SDN mode change protocol (i.e., LO to HI); thereby, it can be done within a short and bounded delay.

B. Autonomous Lane Keeping

The ALK system automatically steers the car to keep it inside the lane. It is considered as a major step toward autonomous driving, as it enables driving without human input. For example, Cadillac's Super Cruise [43] system is a hands-free system that supports ALK with large LIDAR mapping data.

C. Experimental Setup

We evaluate MC-SDN with the ALK system implemented on top of our 1/10 scaled autonomous car presented in Section IX. We implemented the ALK system based on a PID controller which relies on a LIDAR sensor; it senses the surrounding structures and feeds the proper control to keep the car inside the lane.

Fig. 16(a) describes the network topology used in the experiment. We generated two flows: 1) *LIDAR* and 2) *STREAM*, which have the same specification described in Table V. When the link l_1 and l_2 were alive (i.e., normal mode), the LIDAR and STREAM flows passed through the link l_1 and l_2, respectively. In the middle of each experiment, we made a fault at the link l_1 through the interface down command (i.e., `ifdown`). Upon detecting the link fault, each switch changed to the emergency mode with new forwarding rules, so that the LIDAR flow was rerouted to go through l_2 and the STREAM flow was dropped to make room for the LIDAR flow. In order to reflect more complex systems that have tens of flows, each switch has 60 forwarding rules for dummy background flows; upon a mode change, each switch updated all forwarding rules.

D. Experimental Scenario

As shown in Fig. 16(b), the ALK system drove the car while maintaining 0.7 m away from the circular shape wall; we consider this path as the reference lane. We measured the driving trajectory and errors from the lane until the vehicle completes one lap. Note that the car drove itself with an average velocity of 1 m/s, and it takes about 9 s to finish one lap. We made a link fault at 4 s after each experiment started, and evaluated the effectiveness of the fault recovery system. We compare MC-SDN with two baselines: 1) Std-SDN which follows the standard request-response SDN protocol to recover a fault and 2) No-Fault that does not suffer from any link fault and shows the ideal case.

E. Implication

Fig. 16(b) depicts the representative driving trajectory with varying the underlying network system. When the link fault occurs, Std-SDN loses the control and cannot follow the lane, while MC-SDN keeps the control. This difference mainly comes from the expeditious mode change (i.e., fault recovery), by taking advantage of MC-SDN design. Until the fault recovery done, the switch sw_1 transmits the LIDAR flow to the faulted link l_1. Due to the long recovery delay, Std-SDN loses a lot of LIDAR messages. Therefore, the car cannot feed the proper control signal and results in this severe control performance degradation. In contrast, MC-SDN quickly finishes the fault recovery with very few data losses and successfully controls the car inside the lane. Note that the trajectory of MC-SDN is comparable to that of No-Fault.

Fig. 16(c) shows the distributions of the errors, where an error is defined as the distance between the lane and the vehicle position at each control cycle. Note that the car ran the

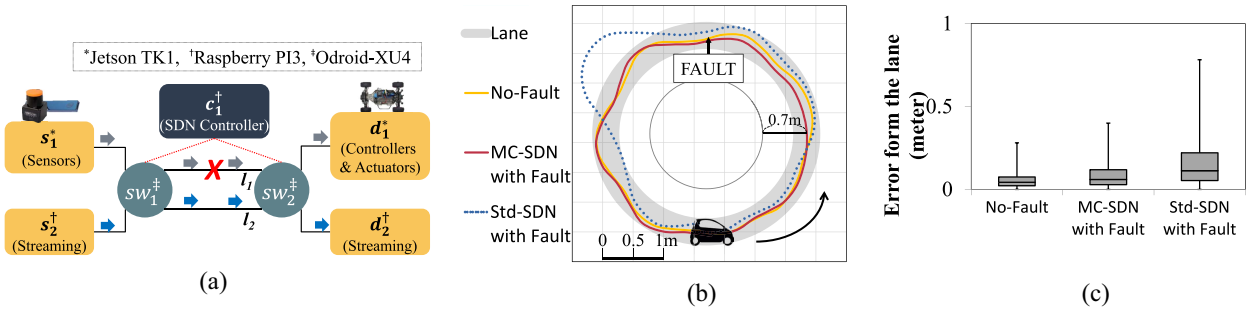


Fig. 16. Link fault recovery performance evaluation with the 1/10 scale autonomous vehicle. (a) Topology: a fault happened at the link l_1. (b) Driving trajectories. (c) Error distributions.

PID control with a frequency of 40 Hz. It shows all measured errors from 20 trials while each gray box covers the 25th–75th percentiles with the line inside indicating the 50th percentile, and the error bar represents the minimum and maximum errors. The smaller the error, the better the control performance. Since No-Fault shows the case without any fault, we refer to No-Fault as an ideal control performance.

In the figure, Std-SDN shows large error values up to 0.78 m, while MC-SDN and No-Fault show the errors up to 0.40 and 0.28 m, respectively. The large error values (up to 0.5 m larger than No-Fault) imply the poor control performance of Std-SDN due to the long fault recovery delay. Note that the errors increase until the fault recovery done. In contrast, MC-SDN results in small errors (up to 0.12 m larger than No-Fault), thanks to the expeditious fault recovery. Note that MC-SDN results in few meters larger errors compared to No-Fault; this is because it takes few milliseconds to detect the link fault. It could be improved by using the well-tuned link fault detection scheme; however, it is out of scope of this paper. Once the link fault is detected, MC-SDN finishes the recovery within less than 1 ms. This fast recovery helps to keep the vehicle under control and results in better control performance.

The evaluation with the new application implies that MC-SDN can be applicable to various systems which require the time-sensitive network rule management. Thanks to the generality of MC-SDN, we believe that it will be a key technology for next-generation cyber-physical networking systems.

XI. Discussion: Supporting More Than Two Criticality Levels

So far in this paper, we have focused on the dual-criticality systems. Thanks to the generality of MC-SDN, it can support systems having more than two criticality levels through some extensions of the internal structure. In this section, we discuss how to extend MC-SDN toward N (>2) criticality levels $L_0 \cdots L_{N-1}$, where L_0 and L_{N-1} denote the lowest and highest criticality levels, respectively.

A. Flow Behavior Monitor With Multiple Requirements

The flow behavior monitor can determine whether a flow is valid in a specific criticality level, based on the multiple requirements for each level. For each message k in a flow τ, the monitor checks the arrival interval and the size of message k with the guide time $G_{\tau,k}$ and the size requirement $C_{\tau,k}$, respectively. Since the flow τ has multiple period requirements $T_\tau(L_i)$, the guide time can be generalized as

$$G_{\tau,k}(L_i) = \max\left\{G_{\tau,k-1}(L_i), E_{\tau,k-1} - J_\tau\right\} + T_\tau(L_i)$$

where $0 \le i < N$. With the generalized guide time $G_{\tau,k}(L_i)$ and multiple size requirements $C_\tau(L_i)$, the monitor can determine the minimum required level of τ $L_{\tau,\text{req}}$, as follows:

$$L_{\tau,\text{req}} = \max\left\{\begin{matrix}\min\left\{L_i | E_{\tau,k} \ge G_{\tau,k}(L_i),\ 0 \le i < N\right\}, \\ \min\left\{L_j | C_{\tau,k} \le C_\tau(L_j),\ 0 \le j < N\right\}\end{matrix}\right\}$$

where $E_{\tau,k}$ and $C_{\tau,k}$ denote the arrival time and size of the message k, respectively. The monitor determines whether the flow τ violates the system mode by comparison between the required level $L_{\tau,\text{req}}$ and the system mode level L_{sys}.

B. Mode Change Protocol

If the monitor sees $L_{\tau,\text{req}} > L_{\text{sys}}$, it determines that the flow τ violates the system mode; the system should change the mode into $L_{\tau,\text{req}}$. When the monitor sees any mode violation, it immediately triggers a mode change toward the higher level. The LO to HI mode change protocol can be directly applicable to this mode change. On the other hand, when the monitor sees $L_{\tau,\text{req}} < L_{\text{sys}}$, it should ensure that all flows in the system are valid in the mode level $L_{\tau,\text{req}}$ for a mode change. The HI to LO mode change protocol is useful for the mode change toward the lower level (refer to Section VII); the flow stats collecting step of the protocol helps to ensure that validity of all other flows in the mode level $L_{\tau,\text{req}}$.

In addition, in order to preserve the system-wide mode level consistency, the mode arranger put the $L_{\tau,\text{req}}$ information into the flow behavior tag when it propagates the mode change signal. According to the mode management principle with the flow behavior tag list, each switch maintains the mode level L_{sys} with the following constraint:

$$\forall_{\tau \in \text{FS}},\ L_{\text{sys}} \ge L_{\tau,\text{req}}$$

where FS denotes the set of flows in the system. Note that MC-SDN can be aware of $L_{\tau,\text{req}}$ information from the flow behavior tag list.

C. MC Rule Manager With Multiple Shadow Tables

The MC rule manager in each switch has multiple shadow tables indexed by the criticality level, and each shadow table contains forwarding rules for the corresponding level. When the system starts, all switches are in L_0 mode level; they use the rules in the shadow table with the index L_0 as their forwarding rules. If a switch is about to change the mode, it refers to the $L_{\tau,\text{req}}$ information in the mode change signal and retrieves new rules from the shadow table indexed as $L_{\tau,\text{req}}$. With this approach, MC-SDN can effectively support N-criticality systems.

XII. Related Work

In this section, we present related works and what differentiates MC-SDN from them.

A. IEEE Time Sensitive Networking

TSN is an Ethernet-based standard suite under development by the IEEE 802.1 TSN Working Group. Since the current bus-based industrial network standards (i.e., CAN [44] and FlexRay [45]) suffer from the bandwidth limitation, there has been an increasing interest in Ethernet as a next-generation standard. TSN defines primitive Ethernet mechanisms to support time-sensitive traffic including: 1) time synchronization [10]; 2) path control and reservation [11]; 3) packet scheduling for various traffic classes [12]–[15]; and 4) packet replication [16]. With these primitives, it supports various classes of time-sensitive (i.e., cyclic/periodic and sporadic) and best-effort traffics in a converged network. Based on the time synchronization [10], TSN provides temporal isolation to time-sensitive traffics by dividing a time interval into multiple time slots and assigning each flow into a dedicated time slot [12], [13].

The nature in packet scheduling is different for TSN and MC-SDN; the former employs only static scheduling, while the latter allows dynamic scheduling as well. For each time-sensitive flow, TSN reserves a fixed amount of network resources statically at design time according to their timing requirement (e.g., a reserved time slot in each cycle), and it is very difficult to change the reservation at runtime. Thus, TSN enforces each flow to make a reservation according to its maximum resource requirements (i.e., its worst case behavior). As an example, let us consider the case of Section IX, where the camera sensor data (HI flow) and streaming (LO flow), respectively, occupy 58% and 88% of bandwidth in the worst case. In this case, TSN cannot make a reservation to guarantee both flows at the same time. One may then think that it is sufficient to make reservation only for HI flows and leave LO flows as best-effort traffics; however, it cannot provide LO flows with any timing guarantees. Note that LO flows (e.g., streaming) are not best-effort, but they are also-time sensitive and require timing guarantee. In contrast, MC-SDN employs mode-based scheduling, which provides strict timing guarantee for HI flows and conditional guarantee (as long as in the LO mode) for LO flows. In case of Section IX, MC-SDN guarantees the timing of all HI and LO flows (i.e., camera sensor and streaming) in the LO mode. In the HI mode, MC-SDN provides timing guarantee for the HI flow while dropping LO flows. Note that although we dropped all LO flows since message loss and delayed delivery are equivalent in terms of real-time communication (i.e., in terms of missing deadlines), MC-SDN also can support LO flows as best-effort (with the lowest priority) in the HI mode. MC-SDN mainly focuses on enabling the mode-based scheduling on Ethernet; it helps to guarantee the timing of MC flows (even LO flows), while demanding significantly less resources than static scheduling.

TSN is orthogonal to MC-SDN. Advanced features of TSN (e.g., time slot reservation) can help to make a more tighter bound of a mode change delay. And, the TSN working group is recently trying to standardize a runtime network management architecture in the P802.1Qcc work-in-progress project [46]. We believe that the fast and predictable network management way proposed by MC-SDN can give a great insight into the upcoming TSN standard, P802.1Qcc.

B. Mixed-Criticality Network Scheduling

Several pieces of research have been studied to support MC flow management (see [3] for a survey) for various networks, including a NoC [5]–[8], CAN [4], and time-triggered Ethernet [9]. It is worthwhile to elaborate the latter work since it considers clock-synchronized switched Ethernet for real-time industrial networks, as we consider switched Ethernet in this paper. The latter work proposes an extension to the IEEE 1588 precision time protocol (PTP) to broadcast a criticality level to all nodes in the network, without considering how to change forwarding rules to drop or re-prioritize packets when the system mode changes. On the other hand, MC-SDN employs in-network MC flow scheduling (through SDN-enabled switches), allowing to handle packets in transit in different ways upon mode changes.

C. SDN for Real-Time Networking

Recently, some studies have proposed SDN approaches for supporting real-time flow scheduling. Qian *et al*. [47] proposed a static routing algorithm to guarantee the timing requirements of real-time messages. Kumar *et al*. [48] proposed a path finding algorithm subject to latency and bandwidth requirement of real-time flows. TSSDN [49] proposes a path finding and time slot allocation algorithm to provides temporal and spatial isolation of real-time flows. MIDAS [50] proposes an admission control based on schedulability test of real-time flows. While the previous studies focus on the control plane algorithms, MC-SDN provides a novel data plane design which enables dynamic network management for MC scheduling.

XIII. Conclusion

This paper presents the design and implementation of MC-SDN that supports MC real-time flows on SDN-based switched Ethernet. It not only presents the first approach to enable a criticality mode change with minimal and bounded delays, based on a deep understanding of SDN, but also provides a sustainable/consistent mechanism supporting LO to HI as well as HI to LO mode changes. We have developed the prototype of MC-SDN not only to examine the performance

closely from various factors, but also to evaluate its effectiveness in a real world system such as a 1/10 scaled autonomous vehicle. The extensive evaluation and case study prove that MC-SDN effectively improves the safety of CPSs.

In this paper, we have implemented MC-SDN on top of a software switch (OVS), to explore the feasibility of supporting real-time MC scheduling on SDN. However, we believe that the design principles of MC-SDN are applicable to general SDN devices, including hardware switches. We leave it future work to optimize the hardware implementation of MC-SDN design with field-programmable gate array-based SDN switches [51].

REFERENCES

[1] AUTOSAR. *Classic Platform Standard 4.3.1*. Accessed: May 20, 2019. [Online]. Available: https://www.autosar.org/standards/classic-platform/classic-platform-431/

[2] *Open Alliance*. Accessed: May 20, 2019. [Online]. Available: http://www.opensig.org/

[3] A. Burns and R. Davis. (2018). *Mixed Criticality Systems—A Review*. [Online]. Available: http://www-users.cs.york.ac.uk/burns/review.pdf

[4] A. Burns and R. I. Davis, "Mixed criticality on controller area network," in *Proc. Euromicro Conf. Real Time Syst. (ECRTS)*, Jul. 2013, pp. 125–134.

[5] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer, "IDAMC: A NoC for mixed criticality systems," in *Proc. IEEE Real Time Embedded Technol. Appl. Symp. (RTAS)*, 2013, pp. 149–156.

[6] A. Burns, J. Harbin, and L. S. Indrusiak, "A wormhole NoC protocol for mixed criticality systems," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, 2014, pp. 184–195.

[7] L. S. Indrusiak, J. Harbin, and A. Burns, "Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip," in *Proc. Euromicro Conf. Real Time Syst. (ECRTS)*, 2015, pp. 47–56.

[8] A. Kostrzewa, S. Saidi, and R. Ernst, "Dynamic control for mixed-critical networks-on-chip," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, 2015, pp. 317–326.

[9] O. Cros, L. George, and X. Li, "A protocol for mixed-criticality management in switched Ethernet networks," in *Proc. Workshop Mixed Critical. Syst. (WMC)*, 2015, pp. 1–6.

[10] *IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Standard 802.1AS, pp. 1–292, Mar. 2011.

[11] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 24: Path Control and Reservation*, IEEE Standard 802.1Qca, pp. 1–120, Mar. 2016.

[12] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 25: Enhancements for Scheduled Traffic*, IEEE Standard 802.1Qbv, pp. 1–57, Mar. 2016.

[13] *IEEE Approved Draft Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks Amendment: Cyclic Queuing and Forwarding*, IEEE Standard P802.1Qch, pp. 1–24, Jan. 2017.

[14] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks—Amendment 26: Frame Preemption*, IEEE Standard 802.1Qbu, pp. 1–52, Aug. 2016.

[15] *IEEE Draft Standard for Local and Metropolitan Area Networks-Bridges and Bridged Networks Asynchronous Traffic Shaping*, IEEE Standard P802.1Qcr, pp. 1–112, Jan. 2018.

[16] *IEEE Standard for Local and Metropolitan Area Networks—Frame Replication and Elimination for Reliability*, IEEE Standard 802.1CB, 2017.

[17] N. Mckeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[18] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, May 2015, pp. 117–130.

[19] *Robot Operating System (ROS)*. Accessed: May 20, 2019. [Online]. Available: http://www.ros.org/

[20] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1215–1227, Nov. 2000.

[21] *Linux Advanced Routing & Traffic Control HOWTO*. Accessed: May 20, 2019. [Online]. Available: http://lartc.org

[22] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[23] *Open Networking Foundation*. Accessed: May 20, 2019. [Online]. Available: https://www.opennetworking.org/

[24] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. USENIX Workshop Hot Topics Manag. Internet Cloud Enterprise Netw. Services (Hot-ICE)*, 2012, p. 10.

[25] Open vSwitch. *An Open Virtual Switch*. Accessed: May 20, 2019. [Online]. Available: http://openvswitch.org/

[26] *Pox, the Python Network Controller*. Accessed: May 20, 2019. [Online]. Available: https://github.com/noxrepo/pox

[27] *Odroid-XU4*. Accessed: May 20, 2019. [Online]. Available: https://magazine.odroid.com/odroid-xu4

[28] *RealTek RTL8152*. Accessed: May 20, 2019. [Online]. Available: https://www.realtek.com/en/products/communications-network-ics/category/10-100m-fast-ethernet

[29] Draka. *SuperCat OUTDOOR CAT 5e U/UTP*. [Online]. Available: https://web.archive.org/web/20120316111058/http://communications.draka.com/sites/eu/Datasheets/SuperCat5_24_U_UTP_Install.pdf

[30] *RTL8152 Datasheet*. Accessed: May 20, 2019. [Online]. Available: https://datasheet.lcsc.com/szlcsc/Realtek-Semicon-RTL8152B-VB-CG_C50656.pdf

[31] R. Wilhelm *et al.*, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 1–53, May 2008.

[32] *BeagleBone Black*. Accessed: May 20, 2019. [Online]. Available: https://beagleboard.org/black

[33] *RealTek RTL8153*. Accessed: May 20, 2019. [Online]. Available: https://www.realtek.com/en/products/communications-network-ics/category/10-100-1000m-gigabit-ethernet

[34] *Jaguar F-PACE ADAS*. Accessed: May 20, 2019. [Online]. Available: https://www.jaguar-me.com/en/jaguar-range/f-pace/features/index.html

[35] NHTSA. *U.S. DOT and IIHS Announce Historic Commitment of 20 Automakers to Make Automatic Emergency Braking Standard on New Vehicles*. Accessed: May 20, 2019. [Online]. Available: https://www.iihs.org/news/detail/u-s-dot-and-iihs-announce-historic-commitment-of-20-automakers-to-make-automatic-emergency-braking-standard-on-new-vehicles

[36] *F1/10 Autonomous Racing Competition*. Accessed: May 20, 2019. [Online]. Available: http://f1tenth.org/

[37] *Traxxas Models*. Accessed: May 20, 2019. [Online]. Available: https://traxxas.com/products/showroom

[38] Hokuyo Automatic Co. *UST-10LX Specification*. Accessed: May 20, 2019. [Online]. Available: https://www.hokuyo-usa.com/products/scanning-laser-rangefinders/ust-10lx

[39] *Intel® RealSense™ Camera R200 DataSheet*. Accessed: May 20, 2019. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/realsense-camera-r200-datasheet.pdf

[40] NVIDIA. *NVIDIA Jetson TK1 Developer Kit Product Page*. Accessed: May 20, 2019. [Online]. Available: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

[41] *Raspberry PI3*. Accessed: May 20, 2019. [Online]. Available: https://www.raspberrypi.org/

[42] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," in *Proc. IEEE Symp. Safety Security Rescue Robot. (SSRR)*, 2011, pp. 155–160.

[43] *Cadillac Super Cruise*. Accessed: May 20, 2019. [Online]. Available: https://www.cadillac.com/world-of-cadillac/innovation/super-cruise

[44] *CAN Specification Version 2.0*, Robert Bosch GmbH, Gerlingen, Germany, Sep. 1991.

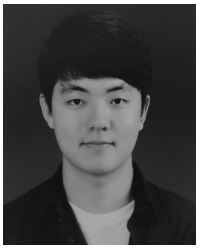
[45] R. Makowitz and C. Temple, "Flexray—A communication network for automotive control systems," in *Proc. IEEE Int. Workshop Factory Commun. Syst.*, 2006, pp. 207–212.

[46] *IEEE Approved Draft Standard for Local and Metropolitan Area Networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks Amendment: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements*, IEEE Standard P802.1Qcc, 2018.

[47] T. Qian, F. Mueller, and Y. Xin, "A linux real-time packet scheduler for reliable static SDN routing," in *Proc. Euromicro Conf. Real Time Syst. (ECRTS)*, 2017, pp. 1–22.
[48] R. Kumar *et al.*, "End-to-end network delay guarantees for real-time systems using SDN," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, 2017, pp. 231–242.
[49] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive software-defined network (TSSDN) for real-time applications," in *Proc. ACM Int. Conf. Real Time Netw. Syst. (RTNS)*, 2016, pp. 193–202.
[50] A. L. King, S. Chen, and I. Lee, "The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow," in *Proc. IEEE Int. Symp. Object Compon. Service Orient. Real Time Distrib. Comput. (ISORC)*, 2014, pp. 133–140.
[51] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep./Oct. 2014.
[52] K. Lee *et al.*, "MC-SDN: Supporting mixed-criticality scheduling on switched-Ethernet using software-defined networking," in *Proc. IEEE Real Time Syst. Symp. (RTSS)*, 2018, pp. 288–299.

Kilho Lee received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2010, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2012 and 2019, respectively.

He is a Post-Doctoral Researcher with the School of Computing, KAIST. His current research interests include system design and implementation for real-time embedded systems and cyber-physical systems.



Minsu Kim received the B.S. degree in computer science and engineering and the B.A. degree in political science and international relations from Seoul National University, Seoul, South Korea, in 2017. He is currently pursuing the M.S. degree in computer science at KAIST, Daejeon, South Korea.

Taejune Park received the B.S. degree in computer engineering from Korea Maritime and Ocean University, Busan, South Korea, in 2013, and the M.S. degree in information security from KAIST, Daejeon, South Korea, in 2015, where he is currently pursuing the Ph.D. degree at the School of Computing.

His current research interests include security issues on software-defined networking/NFV environments and data-planes.

Hoon Sung Chwa (GS'09–M'18) received the B.S., M.S., and Ph.D. degrees from KAIST, Daejeon, South Korea, in 2009, 2011, and 2016, respectively, all in computer science.

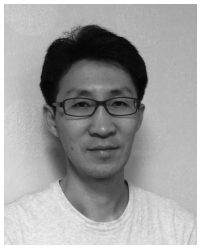
He is an Assistant Professor with the Department of Information and Communication Engineering, DGIST, Daegu, South Korea. He was a Research Fellow with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, until 2018. His current research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems.

Dr. Chwa was a recipient of the Best Paper Award of the 33rd IEEE Real-Time Systems Symposium in 2012 and the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications in 2014.

Jinkyu Lee (GS'07–M'10) received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2004, 2006, and 2011, respectively.

He joined Sungkyunkwan University, Suwon, South Korea, in 2014, where he is an Assistant Professor with the Department of Computer Science and Engineering. He was a Research Fellow/Visiting Scholar with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, until 2014. His current research interests include system design and analysis with timing guarantees, QoS support, and resource management in real-time embedded systems and cyber-physical systems.

Dr. Lee was a recipient of the Best Student Paper Award of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium in 2011 and the Best Paper Award of the 33rd IEEE Real-Time Systems Symposium in 2012.



Seungwon Shin received the B.S. and M.S. degrees in electrical and computer engineering from KAIST, Daejeon, South Korea, and the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University, Uvalde, TX, USA.

He is an Associate Professor with the School of Electrical Engineering, KAIST. Before joining KAIST, he spent nine years within industry, where he devised several mission critical networking systems. He is currently a Research Associate with the Open Networking Foundation, where he is also a member of the Security Working Group. His current research interests include software-defined networking security, Internet of Things security, and botnet analysis/detection.

Insik Shin (M'11) received the B.S. degree from Korea University, Seoul, South Korea, in 1994, the M.S. degree from Stanford University, Stanford, CA, USA, in 1998, and the Ph.D. degree from the University of Pennsylvania, Philadelphia, PA, USA, all in computer science, in 2006.

He joined KAIST, Daejeon, South Korea, in 2008, where he is currently an Associate Professor with the Department of Computer Science. His current research interests include cyber-physical systems and real-time embedded systems.

Dr. Shin was a recipient of the Best Paper Award of the IEEE Real-Time Systems Symposium (RTSS) in 2003 and 2012, the Best Student Paper Award of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2011, and the Best Paper Runner-Up of ECRTS and RTSS in 2008. He is currently an Editorial Board member of the *Journal of Computing Science and Engineering*. He has been the Program Co-Chair of RTCSA and has served various Program Committees in real-time embedded systems, including RTSS, RTAS, and ECRTS.