MC-SDN: Supporting Mixed-Criticality Scheduling on Switched-Ethernet Using Software-Defined Networking

Kilho Lee*, Taejune Park*, Minsu Kim*, Hoon Sung Chwa†, Jinkyu Lee‡, Seungwon Shin§, Insik Shin*
School of Computing, KAIST, Republic of Korea*
Information and Communication Engineering, DGIST, Republic of Korea†
Dept. of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea‡
School of Electrical Engineering, KAIST, Republic of Korea§
Email: insik.shin@cs.kaist.ac.kr

***Abstract*—In this paper, we present the first approach to support mixed-criticality (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 i) 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 ii) 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.**

I. Introduction

Recent advances in embedded systems and communication technologies have led to a growing presence of cyber-physical systems (CPS). CPS generally relies on networks that interconnect sensors, controllers, and actuators to achieve the function of real-time sensing and dynamic control, such as vision-based SLAM (simultaneous localization and mapping) 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].

Another important trend in automotive and avionics industries is towards mixed-criticality (MC) systems that integrate application components with different levels of criticality onto common hardware platforms in order to reduce cost. The scheduling problem of MC systems has been intensively studied in recent years, commonly addressing two seemingly conflicting goals: i) logical separation between applications with different criticality levels and ii) 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 [4], [5]. Following this implication, a few studies investigated the scheduling issue of MC flows on various networks including Controller Area Network (CAN) [6] and Network-on-Chip (NoC) [7]–[10], and the criticality-level management issue on clock-synchronized switches [11]. 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 event-triggered 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, the mode-based MC scheduler should take 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.

Here, we propose to leverage Software-Defined Networking (SDN) for supporting mixed-criticality flows on switched Ethernet, taking advantage of its flexible nature. SDN is an emerging network architecture towards 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 [12], 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

2576-3172/18/$31.00 ©2018 IEEE
DOI 10.1109/RTSS.2018.00045

OpenFlow protocol, 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 FPS 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: i) mode change arrangement, ii) new rule update, and iii) 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 accordingly. This way, MC-SDN eliminates the causes of major delay factors, including OpenFlow communication with the SDN controller, intra-switch 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.

We have implemented a prototype of MC-SDN on top of Open vSwitch [13] 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 show a case study on autonomous driving, where MC-SDN is deployed in a 1/10 scale car[1]. It shows that MC scheduling powered by MC-SDN helps to improve the safety of the driving car in the real world.

The contributions of this paper are summarized as follows:

- Design of a novel mechanism, MC-SDN, to support MC scheduling in switched Ethernet, which is, to the best of our knowledge, the first work that develops mode-based MC scheduling mechanisms on SDN/OpenFlow networks;
- An insightful analysis that shows the limitations of the standard SDN interface (i.e., OpenFlow) when supporting mode change and identifies major delay factors;
- Derivation of a worst-case mode-change delay bound under MC-SDN;
- Evaluation of MC-SDN that reveals orders of magnitude improvement in mode change delays; and
- Demonstration of the effectiveness of MC-SDN, via a case study of a scaled autonomous car, supporting MC flows.

II. System Model and Background

A. System Model

Flow model. We consider a mixed-criticality 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: $<$*period, size, deadline, criticality, source, destination, route*$>$. *Period* is a minimum separation time between two consecutive messages. *Size* is a maximum byte size of each message. *Deadline* specifies a relative end-to-end deadline of each message (Note that *deadline* $\leq$ *period*). *Criticality* is a criticality level. *Source* and *Destination* specify a source node and a destination node (with IP addresses and port numbers), respectively. *Route* is a sequence of nodes that connects a *source* to a *destination*.

Since a message may consist of multiple packets, we consider a network system where end nodes annotate message information on 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 ROS [14], a widely used framework for robot and autonomous driving systems.

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 [15], [16], 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.

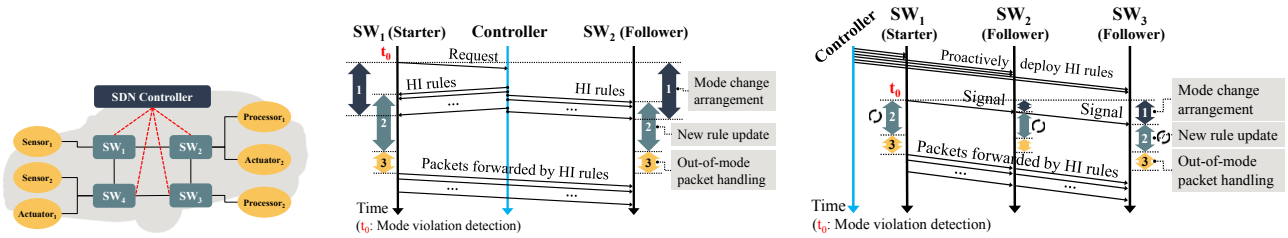
MC scheduling. Like other MC scheduling studies, this paper considers dual-criticality systems having two criticality levels, HI (high) and LO (low), for simplicity [6], [8], [9], [17], [18]. Each HI flow comes with dual requirements for *period* and *size* in HI and LO modes, respectively, while each LO flow has a single requirement for LO mode. 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. The system starts in LO mode. If all flows satisfy 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.

Priority assignment. We assume that the system utilizes *Rate Monotonic* (RM) [19] priority assignment. The shorter the period, the higher the priority. In HI mode, LO flows could be dropped or changed to have the 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

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



(a) MC Networking system overview (b) Std-SDN: Controller-driven mode change (c) MC-SDN: Switch-driven mode change

Fig. 1: System architecture and mode change protocol overview

network packets, to enable dynamic and flexible network control [20]. 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 control and data planes, 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* [12]. 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.

The decoupled structure of SDN brings a high flexibility to network management, and it has a great opportunity to support a mixed-criticality (MC) scheduling which requires highly dynamic packet handling (see Figure 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 policy 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 our proposed MC-SDN design.

A. Motivation Experiment: Controller-Driven Mode Change

We conducted benchmark experiments on a real-world network testbed (refer to Section VII for the testbed details), in order to estimate how long it takes to complete 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 Figure 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 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. A SDN controller is connected to the switches, and it deploys 30 HI rules in mode change. Figure 2 depicts an empirical CDF (Cumulative Distribution Function) 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 FPS in HI mode (i.e., period of 16.7 ms) may miss a deadline due to the 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)

TABLE I: Breakdown of mode change delay

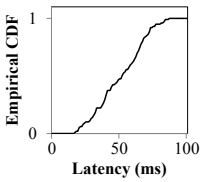


Fig. 2: Empirical CDF of mode change delay

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: i) *mode change arrangement*, ii) *new rule update*, and iii) *out-of-mode packet handling* (see Figure 1(b)). The remainder of this section elaborates delay factors in each step.

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 i) recognizes mode change and ii) receives new rules for HI mode. Such controller-switch communication typically causes a significant delay. According to a measurement study [21], 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, a SDN controller framework, and a 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.

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. The main cause of delay in this step lies in the design structure of SDN switch. A SDN switch typically comprises a number of independent modules following the principle of modular design for performance and management. For example, *switch manager* module receives forwarding rules from the SDN controller through OpenFlow communication, and the forwarding rules are transferred to *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 Open vSwitch, 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). Note that this step can partially overlap with the mode change arrangement step (see Figure 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.

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 Mbps. This delay is very hard to reduce since the link speed depends on the physical constraints.

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 Figure 1(b) to Figure 1(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.

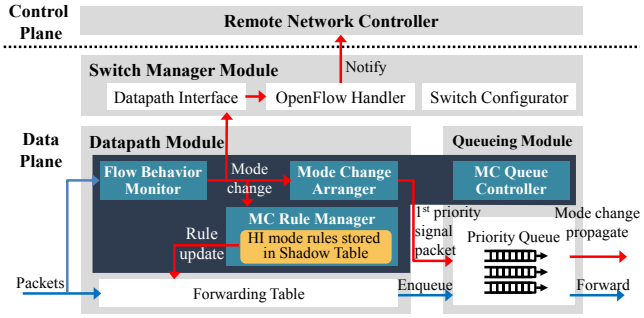


Fig. 3: MC-SDN system architecture

As shown in Figure 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 Figure 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 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 minimizes intra-switch 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, and Section V details implementation issues.

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 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 situation accurately. To this end, we perform traffic pattern analysis based on the *sporadic invariant* with a guide time [6]. 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 i) LO period $-$ release jitter (i.e., $T_\tau(\mathsf{LO}) - J_\tau$), if the last message arrived later than its guide time, or ii) LO period (i.e., $T_\tau(\mathsf{LO})$), if messages are arriving at the maximum rate. Thereby the guide time is calculated as

$$G_{\tau,k}(\mathsf{LO}) = max\{G_{\tau,k-1}(\mathsf{LO}), E_{\tau,k-1} - J_\tau\} + T_\tau(\mathsf{LO}) \text{ [6]}$$

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 [6], 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 the HI mode behavior, it initiates a mode change by utilizing other MC-SDN components presented in the following subsections.

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 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 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 Figure 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 multi-layered 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, fluctuating 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 Open vSwitch), 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, 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 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 the HI mode. The MC queue controller allows to handle out-of-mode packets with 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 Open vSwitch (OVS) [13], [22], which is the de facto standard software switch for OpenFlow implementation.

Target System. We have implemented MC-SDN on top of Open vSwitch (OVS) version 2.4.90, the POX network controller [23], and Linux version 3.10.107. It is worth to describe the internal structure of OVS for ease of understanding this section. As shown in Figure 4, OVS consists of two components, *vswitchd* and *kernel datapath*. The *vswitchd* is a user process in charge of switch management, and it

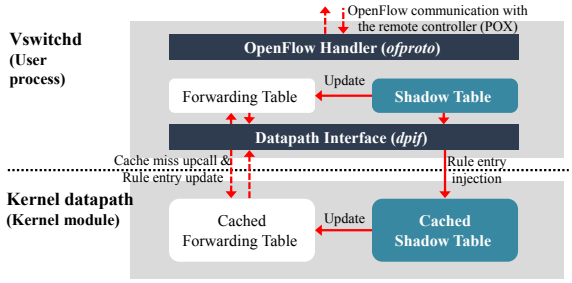


Fig. 4: MC-SDN implementation on Open vSwitch

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 Figure 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 Figure 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 extending *datapath interface (dpif)*; it can inject (or remove) rules into the cached shadow table without the cache miss/update protocol. In addition, we also implemented new OpenFlow commands, `OFPFC_ADD_SHADOW`, `OFPFC_MOD_SHADOW`, and `OFPFC_DEL_SHADOW`, that 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 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 performance and 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; it includes new APIs for POX applications, an extended encoder/decoder in the POX library, extended message handlers and interfaces in *ofproto* and *dpif*, a new data structure in each OVS module, and the functions themselves. Our implementation added around 4,000 and 2,000 lines of code into OVS and POX, respectively.

Flow behavior monitoring is implemented as an OpenFlow action, named `OFPAT_FLOW_MONITOR`. This action is combined with a forwarding action such as `OFPAT_ENQUEUE`; thereby, it can check arrival times and shim-headers of packets before forwarding them. Executing the monitor would impose a delay, but this delay is small enough to be hidden between packet transmissions (See Section VII-C for more details). Mode change arrangement is implemented as another OpenFlow action, named `OFPAT_MODE_ARRANGE`. When detecting a mode violation, `OFPAT_FLOW_MONITOR` calls `OFPAT_MODE_ARRANGE` to trigger mode change. `OFPAT_MODE_ARRANGE` propagates a mode change signal packet that contains a predefined L2 header (i.e., 0x0F00 in `EtherType` field); each switch distinguishes the signal according to that field. It also includes *switch-id* field to avoid broadcast storms by examining redundant signals.

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

During 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*) [16], 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 deqeues 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 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 light weight pointer copy and memory free operations, respectively.

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 VII-C presents experimental results that indicate the MC queue controller imposes a negligible effect on 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.

$$D_{mc} = D_{arrange} + D_{update} + D_{q-handle}, \quad (1)$$

where $D_{arrange}$, D_{update}, and $D_{q-handle}$ 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:

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

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 $D_{q-handle}$ can be expressed as a function of the number of rules to update (N_{rule}) and out-of-mode packets (N_{packet}), respectively:

$$D_{update} = d_{copy} \cdot N_{rule} + d_{u-misc}, \quad (3)$$

$$D_{q-handle} = d_{q-handle} \cdot N_{packet} + d_{q-misc}, \quad (4)$$

where d_{copy} is the maximum required time to copy each rule from the shadow table to the forwarding table; d_{u-misc} is an additional rule-update overhead regardless of N_{rule}; $d_{q-handle}$ is the maximum required time to handle each out-of-mode packet; and d_{q-misc} is an additional overhead regardless of N_{packet}. The additional overheads d_{u-misc} and d_{q-misc} include execution costs to initialize and finalize the iterations, for example, referring internal data structures to access the table and 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 sub-components. Once we upper-bound the worst-case delays of the three components and therefore D_{mc}, we can incorporate D_{mc} into schedulability tests, by adding D_{mc} to the transmission time of HI flows.

B. Upper Bound of Delay Components

In this subsection, we detail how to calculate each delay component in the delay bound, using the real network testbed. The testbed consists of *Odroid-XU4* [24] single board computers equipped with Realtek r8152 [25] USB Ethernet interfaces (see Section VII for details). Note that some delay bounds (e.g., 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.

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 II: Upper-bounds of mode change arrangement delay components

Mode change arrangement. Mode change arrangement delay $D_{arrange}$ consists of 5 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 $530ns$ at 100 meters of Cat.5e UTP Ethernet Cable [26]. We use this value as a safe upper bound, since our system only uses a 1-2 meter-long Ethernet cable. The link transmission delay d_{trans} is calculated as $\frac{packet\ length}{allocated\ bandwidth}$. Since a signal packet has the fixed length of 58 bytes and utilizes full network bandwidth of $100Mbps$ (note that it has the highest priority), d_{trans} can be calculated as $\frac{58*8bits}{100Mbps} = 4.6\mu s$. The queueing delay d_{queue} can be upper-bounded by the transmission time of non-preemptible 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 a 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 [25]) ahead of the signal packet. Considering that the device driver and the NIC hardware can store packets up to 5,460 and 2,048 bytes, respectively [27], d_{queue} can be calculated as $\frac{(5460+2048)*8bits}{100Mbps} = 600.64\mu s$. 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.

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

TABLE III: Upper-bounds of new rule update and out-of-mode packet handling delay components

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, $d_{q-handle}$ 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 of 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 [28]. In addition, the worst-case bound of each component could be tighter through optimizing each mode change step. For example, the order of rule updates could be rearranged by taking into account the underlying system cache structure [29], [30].

VII. Evaluation

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

- How much delay does MC-SDN incur during mode change? (Section VII-A)
- How does the mode change affect end-to-end transmission time under MC-SDN? (Section VII-B).

Experimental setup. Experiments were performed on a network testbed (see Figure 5(a)), which consists of 20 end nodes (*Beaglebone-Black* [31] boards), 9 software switches (*Odroid-XU4* [24] boards), and a SDN controller (A desktop with Intel i5-3750 and 32GB RAM). To increase the connectivity of switch nodes, we equipped each switch node with additional 4 USB Ethernet interfaces (Realtek r8152 [25]) with a USB2.0 hub (Belkin F4U040kr). Switch nodes and end nodes were connected via 100 Mbps Ethernet, and each switch had a dedicated Ethernet interface for the remote SDN controller.

Metrics. 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 NTP (Network Time Protocol) with an accuracy of less than 1 ms.

Mode-based scheduling. Unless stated otherwise, each switch prioritizes packets according to RM (rate-monotonic) 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 Figure 5(b), 5(c), and 5(d), respectively,

A. Mode Change Delay

We ran various experiments to examine how well MC-SDN addresses several delay factors of mode change. During the experiments, we ran a single HI flow that transmits a message of up to 180 KBytes in LO mode with a period of 100 ms. When the HI flow violates its LO mode requirement of message size (i.e, 180 KBytes), a mode change to HI mode occurs.

Figures 6(a) and 6(b) show 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; we will describe the label of *100+* in the following paragraph. The figures show 20 measurements while each gray box covers 25^{th} to 75^{th} percentiles with the line inside indicating 50^{th} percentile, and the error bar represents the minimum and maximum delays. In every scenario, the figures show that Std-SDN incurs significantly larger and highly fluctuating mode change delays than MC-SDN does (note the different scales on the y-axis).

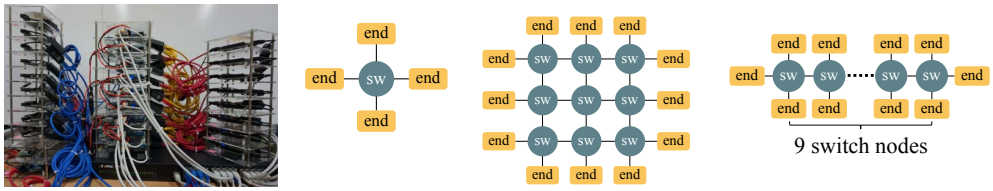
Figure 6(b) also depicts the delay bound as a dotted-line; we calculated the bound with the values presented in Section VI-B. To calculate the bound, we used N_{link} of 0, 4, and 8 according to the topology, N_{packet} of 1000 based on the maximum queue length of network interfaces, and N_{rule} identical 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.

Mode change arrangement. Figure 6 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 switches. In grid and linear topologies, it should propagate up to 4 and 8 hops, and the delay increases as its propagation distance grows[2].

New rule update. Figure 6 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 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 intra-switch cross-layer communication.

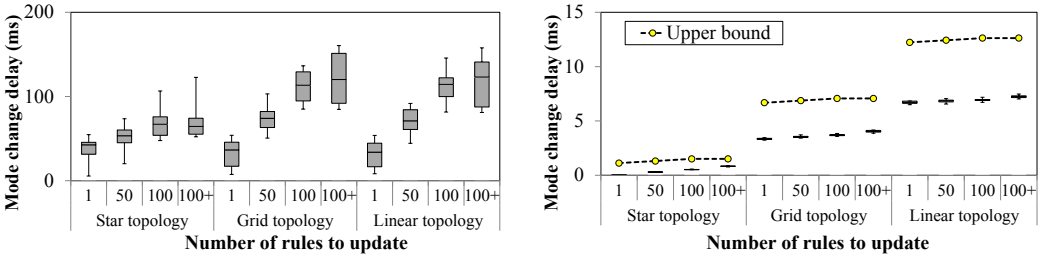
Out-of-mode packet handling. To evaluate the out-of-mode packet handling, we ran experiments with two additional LO flows that share the links with the HI flow, where 100 rules are updated in mode change. Each LO flow has the period of 100 ms and the size of 490 Kbytes. Figure 6 shows the results on the x-axis labeled *100+*. 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). This is because MC-SDN immediately drops out-of-mode packets (i.e., LO flows) out of the queue, while Std-SDN transmits those LO flows.

[2]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.



(a) Testbed picture (b) Star topology (c) Grid topology (d) Linear topology

Fig. 5: Network testbed and various topology



(a) Std-SDN (b) MC-SDN

Fig. 6: Mode change delays with varying the topology, the number of rules to update, and the presence of out-of-mode packets

B. End-to-End Transmission Time

In this subsection, we evaluate the effect of mode change on end-to-end transmission time over various experiment scenarios on the grid topology depicted in Figure 5(c).

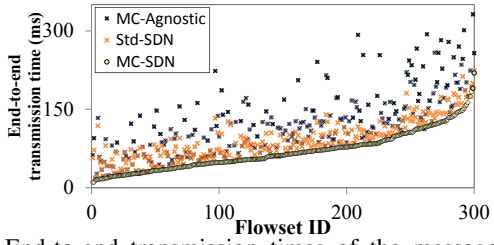


Fig. 7: End-to-end transmission times of the messages that show HI behavior at the first time

In each experiment, we generated a set of up to 16 real-time flows as follows. For each flow, we selected its period randomly between 10 ms and 200 ms, a message *size* that can be transmitted randomly between 10% and 40% of the period on a 100 Mbps link, and a *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 Figure 7 plots the end-to-end transmission times of the message M on MC-SDN, Std-SDN, and MC-Agnostic, respectively, in an increasing order of MC-SDN's measurements for the ease of presentation.

As shown in Figure 7, 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 ms and 20 ms longer end-to-end transmission times than MC-SDN, respectively; in the worst case, they respectively show 211 ms and 102 ms longer results than MC-SDN. 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 Figure 7. 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, Figure 7 implies that MC-SDN effectively reduces the mode change delay and then improves schedulability of MC flows, in general cases.

C. Packet Forwarding Overhead

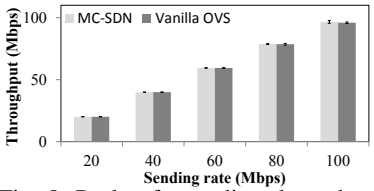


Fig. 8: Packet forwarding throughput

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 subsection, we measured the overhead of MC-SDN with the star topology testbed shown in Figure 5(b). An end node generated messages with varying the sending rate; the MC-SDN switch monitored the flow behavior, and another end node measured the forwarding throughput.

In Figure 8, each bar graph with its error bar represents the average overhead with the standard deviation of 20 trials, and each trial ran for five seconds. We compare the result with *vanilla OVS*, an unmodified version of OVS. The figure shows that the throughput of MC-SDN is comparable to that of vanilla OVS regardless of the sending rate. This is because the action only incurs very short delay (i.e., the average of 760 *ns* per packet) that can be effectively hidden in between packet transmissions. Besides, other components of MC-SDN

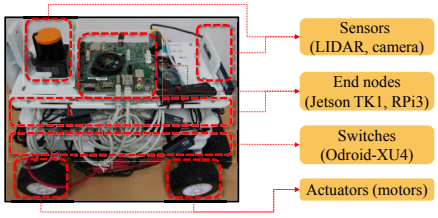


Fig. 9: 1/10 scale autonomous vehicle

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

VIII. Case Study: Autonomous Vehicle

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

Autonomous Emergency Braking. AEB brakes the car in emergency situations by predicting the risk of collisions with detecting obstacles through various sensors. Since AEB effectively lowers the risk of accidents, automakers have agreed to equip it as a standard feature [32]. For instance, Jaguar F-PACE equips with the depth camera based AEB [33].

Experiment setup. We have implemented a 1/10 scaled autonomous car, as shown in Figure 9, which is extended from the F1/10 autonomous racing platform [34]. The car consisted of the Traxxas Rally 1/10 body with actuators (i.e., motors) [35], sensors including a LIDAR (Hokuyo UST-10LX [36]) and a depth camera (Intel Realsense R200 [37]), Jetson TK1 [38] and Raspberry PI3 [39] boards for end nodes, Odroid-XU4 [24] boards for switch nodes, and an additional Raspberry PI3 for a SDN controller. As shown in Figure 10(a), each node was connected with each other via 100 Mbps 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 *Simultaneous Localization And Mapping* (SLAM) [40] and the PID controller for motors. They were implemented as components of the *Robot Operating Systems* (ROS) [14] framework.

Flow	Period (ms)		Priority		Size (KB)	Src.	Dst.
	LO	HI	LO	HI			
LIDAR	40	40	high	high	8	s_1	d_1
STREAM	40	40	mid	drop	432	s_2	d_2
CAM	200	22	low	low	154	s_1	d_1

TABLE IV: Flow set specification in the car system

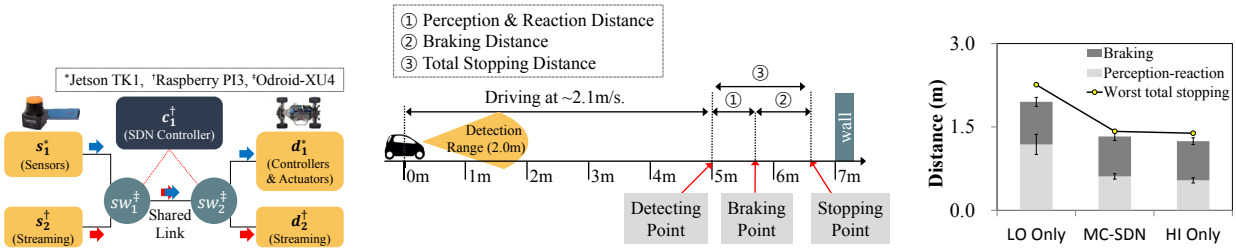
As shown in Table IV, the car system generated three real-time flows: i) *LIDAR* for the LIDAR sensor data used for SLAM, ii) *STREAM* for video frames of user entertainment, and iii) *CAM* for image frames from the depth camera used for AEB. Note that the period of the CAM flow had multiple requirements 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 IV; 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).

Experiment scenario. Figure 10(b) illustrates the experiment scenario. The car was placed 7 meters ahead of a wall, and it drove itself towards the wall at the top speed of 2.1 m/s. We restricted the camera's field of view to 2 meters, 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 & reaction*, *braking*, and *total stopping* distances, respectively, as depicted in Figure 10(b) and use them as performance metrics. To show the effectiveness of MC-SDN, we used two baselines: *LO Only* and *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 IV. 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.

Implication. Figure 10(c) depicts the braking performance of the car with varying 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 & 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 the 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 ms 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. At the same time, 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 the system to balance between two conflicting objectives: efficient resource sharing and safety-critical real-time requirements guarantee.



(a) System architecture and topology (b) Evaluation overview (c) Stopping distances

Fig. 10: Evaluation with the 1/10 scale autonomous vehicle

IX. Related Work

CPS network standard. Nowadays, several network standards for cyber physical systems have been established. Control Area Network (CAN) [41] and FlexRay [42] are industrial standards widely used for automobiles. They construct bus-based networks, and ensure reliable and deterministic data transmission. Despite the reliability, the limited bandwidth of them has led to Ethernet as a next-generation standard. Avionics Full-Duplex Switched Ethernet (AFDX) [43] and BroadR-Reach [2] are switched Ethernet based network standards for avionics and automotive systems; they provide physical layer standards for high bandwidth full duplex control networks. Beyond the reliability of the standards, MC-SDN enables a dynamic real-time flow management, which could be a key property to improve control performance and user experience of cyber-physical systems.

IEEE TSN Working Group. Time-Sensitive Networking (TSN) is a standard suite under development by the IEEE 802.1 TSN Working Group. It defines Ethernet mechanisms for time-sensitive traffic transmission including time synchronization [44], path control and reservation [45], traffic scheduling [46], and frame preemption [47]. They present key technologies to enable more reliable packet transmission with a small and predictable latency. However, TSN cannot support the fast and predictable mode-based rule management, the key idea of MC-SDN. Instead, TSN can be complementary to MC-SDN. For instance, it can provide a tighter mode change delay bound based on its advanced packet handling mechanisms.

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 Network-on-Chip (NoC) [7]–[10], Controller Area Network (CAN) [6], and clock-synchronized switched Ethernet [11]. 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 IEEE 1588 PTP (Precision Time Protocol) 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.

SDN for real-time networking. Recently, some studies have proposed SDN approaches for supporting real-time flow scheduling. Qian *et al.* [48] proposes a static routing algorithm to guarantee timing requirements of real-time messages. Kumar *et al.* [49] proposes a path finding algorithm subject to latency and bandwidth requirement of real-time flows. TSSDN [50] proposes a path finding and time slot allocation algorithm to provides temporal and spatial isolation of real-time flows. MIDAS [51] 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 a dynamic network management for MC scheduling.

X. Conclusion and Discussion

This paper presents the design and implementation of MC-SDN that supports mixed-criticality real-time flows on SDN-based switched Ethernet. It presents the first approach to enable a criticality mode change with minimal and bounded delays, based on a deep understanding of SDN. 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 cyber-physical systems.

In this paper, we focused on identifying and addressing delay factors in mode change, while considering a mode violation in dual-criticality systems (i.e., LO to HI mode change). Yet, the network should be able to go back to LO mode for system sustainability. The mode change in the opposite direction (HI to LO) has slightly different requirements; it raises several challenges such as determining the safe mode change timing and maintaining the system-wide mode consistency. In addition, for real systems which require a complicated level of assurance [52], MC-SDN should be generalized towards supporting more than two criticality levels. It also raises additional challenges, including managing the system mode and forwarding rules according to the multiple criticality levels. We leave them as future work.

Furthermore, we have implemented MC-SDN on top of a software switch (Open vSwitch), 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 FPGA-based SDN switches [53].

ACKNOWLEDGEMENTS

This work was supported in part by BSRP (NRF-2015R1D1A1A01058713), IITP (2014-0-00065, Resilient Cyber-Physical Systems Research), DAPA/ADD (High-Speed Vehicle Research Center of KAIST, UD170018CD), ERC (NRF-2018R1A5A1059921), MSIT (DGIST Start-up Fund Program, 2018080005), MSIT (NRF-2017R1A2B2002458), and MOLIT (17TBIP-C125224-01).

REFERENCES

[1] "Autosar classic platform standard 4.3.0," 2016.
[2] "Open alliance," "http://www.opensig.org/".
[3] A. Burns and R. Davis, "Mixed criticality systems – a review," 2017, http://www-users.cs.york.ac.uk/burns/review.pdf, the ninth edition.
[4] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: Fluid model-based mixed-criticality scheduling on multiprocessors," in *RTSS*, 2014.
[5] H. Baek, N. Jung, H. S. Chwa, I. Shin, and J. Lee, "Non-preemptive scheduling for mixed-criticality real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1766–1779, Aug. 2018.
[6] A. Burns *et al.*, "Mixed criticality on controller area network," in *ECRTS*, 2013.
[7] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer, "Idamc: A noc for mixed criticality systems," in *RTAS*, 2013.
[8] A. Burns, J. Harbin, and L. S. Indrusiak, "A wormhole noc protocol for mixed criticality systems," in *RTSS*, 2014.
[9] L. S. Indrusiak, J. Harbin, and A. Burns, "Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip," in *ECRTS*, 2015.
[10] A. Kostrzewa, S. Saidi, and R. Ernst, "Dynamic control for mixed-criticality networks-on-chip," in *RTSS*, 2015.
[11] O. Cros, L. George, and X. Li, "A protocol for mixed-criticality management in switched ethernet networks," in *Workshop on Mixed Criticality Systems (WMC)*, 2015.
[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.
[13] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *NSDI*, 2015.
[14] Robot Operating System (ROS), http://www.ros.org/.
[15] S.-W. Moon, K. G. Shin, and J. Rexford, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computer*, vol. 49, no. 11, pp. 1215–1227, Nov. 2000.
[16] "Linux Advanced Routing & Traffic Control HOWTO," http://lartc.org.
[17] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee, "MC-ADAPT: Adaptive task dropping in mixed-criticality scheduling," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 163:1–163:21, Sep. 2017.
[18] H. S. Chwa, K. G. Shin, H. Baek, and J. Lee, "Physical-state-aware dynamic slack management for mixed-criticality systems," in *RTAS*, 2018.
[19] 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.
[20] ONF, "Open Networking Foundation," www.opennetworking.org/.
[21] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Hot-ICE*, 2012.
[22] Open vSwitch, "An Open Virtual Switch," http://openvswitch.org/.
[23] "Pox, the python network controller," https://github.com/noxrepo/pox.
[24] "Odroid-XU4," https://magazine.odroid.com/odroid-xu4.
[25] "RealTek RTL8152," http://www.realtek.com.tw/products/productsView.aspx?Langid=1&PNid=14&PFid=55&Level=5&Conn=4&ProdID=323.
[26] Draka, "SuperCat OUTDOOR CAT 5e U/UTP," https://web.archive.org/web/20120316111058/http://communications.draka.com/sites/eu/Datasheets/SuperCat5_24_U_UTP_Install.pdf.
[27] RTL8152 Datasheet, https://datasheet.lcsc.com/szlcsc/Realtek-Semicon-RTL8152B-VB-CG_C50656.pdf.
[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.
[29] W. Chang, D. Goswami, S. Chakraborty, L. Ju, C. J. Xue, and S. Andalam, "Memory-aware embedded control systems design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 586–599, 2017.
[30] C. Wanli and S. Chakraborty, "Resource-aware automotive control systems design: A cyber-physical systems approach," vol. 10, pp. 249–369, 01 2016.
[31] "BeagleBone Black," https://beagleboard.org/black.
[32] NHTSA, "U.S. DOT and IIHS announce historic commitment of 20 automakers to make automatic emergency braking standard on new vehicles," https://www.nhtsa.gov/press-releases/us-dot-and-iihs-announce-historic-commitment-20-automakers-make-automatic-emergency.
[33] Jaguar F-PACE ADAS, https://www.jaguar-me.com/en/about-jaguar/jaguar-stories/f-pace-advanced-driver-assistance-systems.html.
[34] F1/10 Autonomous Racing Competition, http://f1tenth.org/.
[35] Traxxas Models, https://traxxas.com/products/showroom.
[36] Hokuyo Automatic Co., "UST-10LX Specification," http://www.senteksolutions.com/application/files/2414/7196/1936/UST-10LX_Specifications.pdf.
[37] Intel, "Intel® RealSense™ Camera R200 product datasheet," https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/realsense-camera-r200-datasheet.pdf.
[38] NVIDIA, "NVIDIA Jetson TK1 developer kit product page," http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html.
[39] Raspberry PI3, https://www.raspberrypi.org/products/raspberry-pi-3-model-b.
[40] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011.
[41] R. Bosch, "CAN specification version 2.0," Sep. 1991.
[42] R. Makowitz and C. Temple, "Flexray-a communication network for automotive control systems," in *IEEE International Workshop on Factory Communication Systems*, 2006, pp. 207–212.
[43] J.-P. Moreaux, "Data transmission system for aircraft," Aug. 2 2005, uS Patent 6,925,088.
[44] "IEEE standard for local and metropolitan area networks - timing and synchronization for time-sensitive applications in bridged local area networks," *IEEE Std 802.1AS-2011*, pp. 1–292, March 2011.
[45] "IEEE standard for local and metropolitan area networks– bridges and bridged networks - amendment 24: Path control and reservation," *IEEE Std 802.1Qca-2015*, pp. 1–120, March 2016.
[46] "IEEE standard for local and metropolitan area networks – bridges and bridged networks - amendment 25: Enhancements for scheduled traffic," *IEEE Std 802.1Qbv-2015*, pp. 1–57, March 2016.
[47] "IEEE standard for local and metropolitan area networks – bridges and bridged networks – amendment 26: Frame preemption," *IEEE Std 802.1Qbu-2016*, pp. 1–52, Aug 2016.
[48] T. Qian, F. Mueller, and Y. Xin, "A linux real-time packet scheduler for reliable static sdn routing," in *ECRTS*, 2017.
[49] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. Bobba, "End-to-end network delay guarantees for real-time systems using sdn," in *RTSS*, 2017.
[50] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive software-defined network (tssdn) for real-time applications," in *RTNS*, 2016.
[51] A. L. King, S. Chen, and I. Lee, "The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow," in *ISORC*, 2014.
[52] ISO, "ISO 26262: Road vehicles - Functional safety - Part 9: Automotive Safety Integrity Level (ASIL)," 2011.
[53] 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, 2014.