A Linux Real-Time Packet Scheduler for Reliable Static SDN Routing*†

Tao Qian[1], Frank Mueller[2], and Yufeng Xin[3]

1 **North Carolina State University, Raleigh, NC, USA**
`tqian2@ncsu.edu`
2 **North Carolina State University, Raleigh, NC, USA**
`mueller@cs.ncsu.edu`
3 **RENCI, The University of North Carolina, Chapel Hill, NC, USA**
`yxin@renci.org`

Abstract

In a distributed computing environment, guaranteeing the hard deadline for real-time messages is essential to ensure schedulability of real-time tasks. Since capabilities of the shared resources for transmission are limited, e.g., the buffer size is limited on network devices, it becomes a challenge to design an effective and feasible resource sharing policy based on both the demand of real-time packet transmissions and the limitation of resource capabilities. We address this challenge in two cooperative mechanisms. First, we design a static routing algorithm to find forwarding paths for packets to guarantee their hard deadlines. The routing algorithm employs a validation-based backtracking procedure capable of deriving the demand of a set of real-time packets on each shared network device, and it checks whether this demand can be met on the device. Second, we design a packet scheduler that runs on network devices to transmit messages according to our routing requirements. We implement these mechanisms on virtual software-defined network (SDN) switches and evaluate them on real hardware in a local cluster to demonstrate the feasibility and effectiveness of our routing algorithm and packet scheduler.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases real-time networks, packet scheduling, deadline guarantee

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.25

1 Introduction

In a distributed computing environment, multiple compute nodes share communication resources to transmit data in order to collaborate with each other. In such systems, employing an effective resource sharing mechanism is essential to meet the real-time requirements of tasks. These mechanisms can be divided into two categories. First, the compute nodes control their own behavior of how to utilize shared resources. Past research has studied mechanisms of shaping the resource access pattern to increase timing predictability, e.g., memory sharing based on limiting the memory bandwidth for different cores [31, 32] and network bandwidth limitation on compute nodes connected via Ethernet [23]. These mechanisms are *passive* since they can only reduce the probability of resource contention instead of preventing contention in the first place. Thus, passive mechanisms usually guarantee probabilistic deadlines. The second category includes *active* resource sharing mechanisms, which either assign the resource

* This work was supported in part by NSF grants 1525609, 1329780, 1239246, 0905181 and 0958311.
† Aranya Chakrabortty helped to scope the problem in discussions.

© Tao Qian, Frank Mueller, and Yufeng Xin;
licensed under Creative Commons License CC-BY

to exactly one task at a time to prevent contention (e.g., TDMA-based bus sharing on a multiprocessor platform [25]), or provide mechanisms on the shared resources to control its usage directly [1, 14]. One benefit of active mechanisms is that by controlling how resources are shared, one can build up a mathematical model to estimate the resource access time. We believe this model is a necessary condition to guarantee hard deadlines.

In our previous work, we implemented a cyclic executive based task scheduler on distributed nodes to guarantee probabilistic task deadlines [21] and a hybrid earliest-deadline-first (EDF) task and packet scheduler to guarantee hard deadlines [23]. Furthermore, we have implemented a real-time distributed hash table (RT-DHT) to support the scalability and resilience requirements of distributed wide-area measurement systems, which estimate power grid oscillation modes based on real-time power state data [22, 19]. A passive mechanism (bandwidth limitation on DHT nodes) was adopted to reduce variations of the network delay in order to increase the schedulability of our hybrid scheduler [23]. However, when the RT-DHT has to share network resources with other systems whose network utilization patterns are unpredictable, this passive mechanism is not enough since it cannot restrict the network access of other systems. In this work, we actively control network devices (e.g., switches) by enforcing a packet scheduling algorithm on the devices to guarantee hard message transmission deadlines for real-time packets, even when these devices are subject to unpredictable background traffic.

One of the challenges of adopting the active mechanism is to determine an accurate resource sharing policy that considers both the demands of the real-time tasks and the capacities of the shared resources. For example, an active memory controller needs to know the bandwidth demands of the running tasks to proactively reserve bandwidth for each of them. This challenge becomes even harder for a network since the topology of the network can be complex (e.g., network devices have to collaborate in order to guarantee the deadline). We need to derive an effective static routing algorithm to determine the forwarding paths for all real-time packets so that the end-to-end delay for such a packet through the corresponding path never exceeds its deadline. Without loss of generality, we use term *packet* and *message* interchangeably in this paper.

One possible approach to address this problem is to express it as an integer linear programming (ILP) problem if the objective function and constraints are linear. In our case, each real-time message could have multiple forwarding paths with different costs for each path. For example, one cost is the size of the buffer the message will occupy on every network device along the path. Then the problem becomes to assign a forwarding path for each real-time message under the condition that the constraints on each shared device can be met (in this case, the total size of messages is no more than the buffer size of the shared device at any time). The effectiveness of this model is based on the accuracy of the cost function. However, for the routing problem, the cost function on each device is dependent on the exact traffic that goes through that device (i.e., dependent on the assignment results). This cyclic dependency makes this a hard problem. Previous work has proposed to utilize different oracles to break the cyclic dependency [12]. For example, assuming an oracle that determines the transmission time on any device for any message at any time exists, the assignment problem then can be solved with ILP techniques. In this paper, we enforce an active message scheduling algorithm on network devices so we can derive the cost function for each message and each device on the path for all traffic that goes through the device. As a result, we use a validation-based backtracking algorithm to determine the forwarding path for each message (detailed in Section 2.3).

The objective of our scheduling algorithm is to avoid dropping real-time messages even when network congestion occurs, e.g., due to too many real-time messages and background

traffic (considered as non real-time). First, the routing algorithm needs to guarantee that when multiple real-time messages share a device on their forwarding paths, the aggregate message size does not exceed the buffer size of that device. Thus, the scheduling algorithm needs to control the timing when a message is transmitted from one device to the next. The transmission time is planned statically so that the aggregate size of real-time messages on a device can be calculated in our static routing algorithm. Second, the scheduling algorithm needs to drop background traffic when the available buffer on a device is insufficient for real-time messages. In this way, our scheduling algorithm guarantees real-time message deadlines while providing a best-effort service to background traffic.

In summary, the contributions of this work are: (1) We design a static routing algorithm to find forwarding paths for multiple real-time messages to guarantee the hard deadlines of messages. (2) We propose a deadline-based scheduling algorithm, which actively controls the time at which real-time messages are processed at each device and only drops background packets when the device is congested. (3) We implement the packet scheduler on virtual SDN switches and conduct experiments to evaluate our approach.

The rest of paper is organized as follows. Section 2 presents the design of our static routing algorithm and the active device scheduling algorithm. Section 3 presents the implementation details of the scheduling algorithm on virtual switches. Section 4 discusses the evaluation setup and results. Section 5 contrasts our work with related work. Section 6 presents the conclusion and on-going research.

2 Design

This section first presents the system model and the objectives. It then contributes the static routing algorithm that determines forwarding paths for multiple pairs of source and destination of real-time messages. After that, it contributes the scheduling algorithm that runs on network devices to guarantee that real-time messages are transmitted in a time predictable fashion.

2.1 System Model and Objectives

Network Model: We use notation (V, E, B, Pr) to represent the underlying network utilized by compute nodes in distributed real-time systems to exchange messages. V denotes a finite set of compute nodes and the networking hardware, which forwards messages between compute nodes. Without loss of generality, we use the term *node* to represent either a networking device or a compute node. Let B be buffer sizes on nodes, i.e., the aggregate size of messages that can be stored in the queue on each node. If a burst of messages arriving at a node exceeds the buffer size, a fraction of these messages will be dropped. Thus, one objective of our system is to guarantee that only background traffic (i.e., messages without deadline requirements) can be dropped in such a case. Matrix E represents the real-time links between nodes in the graph. Nodes v_1 and v_2 have a physical connection for real-time traffic iff $E[v_1, v_2] > 0$, where $E[v_1, v_2]$ is the interface speed. Matrix Pr represents the propagation delays on these links. Table 1 summarizes the notation. When a network runs in stable state, its nodes and links do not change over time. Thus, V, E, B, and Pr are constant. We require that clock times on nodes are synchronized. This can be achieved via the Network Time Protocol [18], running at system startup time and periodically as a real-time task, or hardware support (e.g., GPS of phasor measurement units in the power grid).

Table 1 Notation.

Name	Meaning
V	set of nodes
$B[v]$	buffer size on node v
$E[v_1, v_2]$	constant interface speed matrix for real-time traffic between nodes v_1 and v_2
$Pr[v_1, v_2]$	constant propagation delay matrix on links between v_1 and v_2
D_m	relative deadlines of a message flow m
T_m	periods of a message flow m
S_m	size of messages in flow m
$R[v]$	expected message response time on node v
$A[v]$	latest message transmission time on node v
$\delta[v]$	runtime message response time variance on node v
$\Delta[v]$	response time variance bound on node v, determined by all message flows on the node

Node Architecture: We consider a store-and-forward node model. A packet arriving at a node is first put into the input buffer. Then, the node processor (i.e., forwarding engine) processes the packet to determine the forwarding rule for that packet (e.g., time to forward the packet and output link) and forwards the packet to the corresponding output queue at scheduled time (e.g., Cisco Calalyst Switches [6]). This is further detailed in Section 2.2.

Message Release Model: We consider two types of messages. First are messages due to background traffic without deadlines. Second are real-time messages released periodically by real-time tasks running on one end node, which are subsequently transmitted to another node. These messages can be classified as message flows, where a flow contains all messages released by the same real-time task. Thus, messages in the same flow have the same source and destination nodes, same relative deadline, and are released periodically. We define a set of message flows as $M = (D, T, S)$, where D is the relative deadlines of the message flows, T is the periods, and S is the sizes. The flow id and the release time of a message are embedded in the message header upon transmission. Nodes use the flow id to look up the forwarding policy for that flow from their routing table and use the release time to calculate the exact forwarding time for a specific message in that flow.

Objectives: Given the configuration of real-time flows, our static routing algorithm shall derive a forwarding path for each flow, such that (1) the transmission time of a real-time message shall never exceed its relative deadline, (2) the aggregate size of real-time messages on any network device shall never exceed the buffer size of that device, (3) when multiple real-time messages share a network device on their paths, the network device shall have the computing capability to process them before their local deadlines. To achieve these goals, the static routing algorithm derives offline the expected response time R by which each node must have processed a real-time message. The goal for the scheduler is to guarantee end-to-end message deadlines by enforcing the expected response time on each node and considering the runtime response time variance δ caused by the interference due to scheduling and queueing at the output interface. Section 2.3 proves that the scheduler can adopt an EDF-based algorithm that uses R as local deadlines to achieve this goal.

Message Delay Model: We focus on real-time messages to be transmitted before their deadlines. Thus, given any real-time message flow, the objective is to find a forwarding path for which the end-to-end delay does not exceed the relative deadline for the flow. More formally, let the forwarding path for a flow be $(v_0, v_1, ..., v_k)$, where k is the path length and $v_i \in V(0 \leq i \leq k)$ are the nodes on the forwarding path. Let $t[v_i]$ be the time when the message arrives at node v_i. $t[v_0]$ is the message release time, which is determined by the property of the corresponding message flow. $t[v_i]$, $0 < i \leq k$, can be formalized as a function of the path and the underlying network as shown in Eq. 1, where $R[v_i]$ is the expected message response time on node v_i and $\delta[v_i]$ is the runtime variance for the message response time (i.e., node v_i starts to transmit the message at time $t[v_i] + R[v_i]$ and finishes the transmission at time $t[v_i] + R[v_i] + \delta[v_i]$). The first sum in Eq. 1 is the accumulated propagation delay on links from v_0 to v_i. The second sum is the accumulated message response time on nodes v_0 to v_{i-1}. The subscript m for the message flow is abbreviated when the equations are suitable for every message flow (e.g., $R[v_j]$ instead of $R_m[v_j]$).

$$\begin{aligned} t[v_i] &= t[v_{i-1}] + R[v_{i-1}] + \delta[v_{i-1}] + Pr[v_{i-1}, v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i-1} (R[v_j] + \delta[v_j]). \end{aligned} \tag{1}$$

Let $A[v_i]$ be the worst case (i.e., the latest time) that node v_i has to start the transmission. On the first node, $A[v_0] = t[v_0] + R[v_0]$. We assume that $\delta[v_i] \geq 0$ on any node v_i. $\delta[v_i]$ can be bounded by a value $\Delta[v_i]$, which is determined by all message flows on node v_i and is the same for any individual message on this node. Then, the latest transmission time $A[v_i]$ $(0 < i \leq k)$ occurs when the message transmissions on all nodes before v_i have experienced the worst variation, which is expressed in Eq. 2. Thus, $A[v_i]$ can be calculated for a message once R and Δ are derived. Sections 2.2 and 2.3 detail the approach to derive R and Δ offline.

$$\begin{aligned} A[v_i] &= max\{t[v_i] + R[v_i]\} = max\{t[v_i]\} + R[v_i] \\ &= t[v_0] + \sum_{j=0}^{i-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{i} R[v_j] + \sum_{j=0}^{i-1} \Delta[v_j]\,. \end{aligned} \tag{2}$$

As a result, the static routing algorithm needs to find a forwarding path for every message flow so that the end-to-end delay in the worst case is bounded by its relative deadline as expressed in Eq. 3.

$$A[v_k] + \Delta[v_k] - t[v_0] \leq D\,. \tag{3}$$

2.2 Message Scheduler

A real-time message experiences three phases on one node. Fig. 1 illustrates these phases on node v, where the message is sent by node v' and received by node v, and then forwarded to node v''. Before the message is sent to node v, the processor on node v' has calculated the latest message transmission time for the next node (i.e., $A[v]$) from the forwarding table. In the first phase, the message arrives from link 1 at time $t[v]$ and is put in the input buffer. We assume that the time for an interface to put a packet into the input buffer can be ignored. In the second phase, the message is processed and moved to the intermediate queue. During processing, the latest message transmission time for the next node (i.e., $A[v'']$) is calculated and the output interface for the message is determined according to the forwarding table.

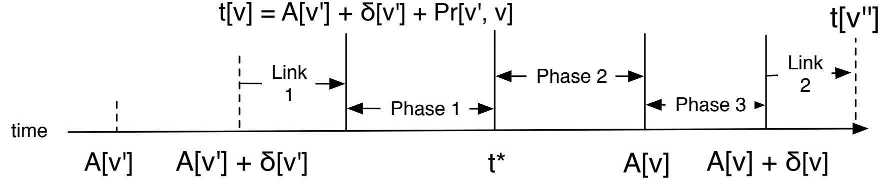

Figure 1 Message Phases on Node.

The message stays in the intermediate queue during the second phase. In the third phase, the message is forwarded to the corresponding interface at time $A[v]$ and then finally transmitted via link 2 at time $A[v] + \delta[v]$. The intuition of this design is to constrain the message arrival time at each node v to a time interval $(A[v'] + Pr[v', v], A[v'] + Pr[v', v] + \Delta[v'])$. In contrast, a background traffic packet is moved from the input buffer to the corresponding interface directly when no real-time messages need to be processed.

To guarantee that the processor can start the transmission in the third phase at time $A[v]$, the processor has to finish processing the message before $A[v]$ in the second phase (at time t^* in Fig. 1). Thus, our packet scheduler adopts an EDF-based algorithm to process the message before its local relative deadline $A[v] - t[v]$. Since $A[v] = A[v'] + Pr[v', v] + \Delta[v'] + R[v]$, which is derived from Eq. 2, the local relative deadline $A[v] - t[v]$ is no less than the expected response time $R[v]$ as derived in Eq. 4. If the processor has the computation capability to process every real-time message before its expected response time $R[v]$, the processor can forward it on time.

$$\begin{aligned} A[v] - t[v] &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) - t[v] \\ &= (A[v'] + Pr[v', v] + \Delta[v'] + R[v]) \\ &\quad - (A[v'] + Pr[v', v] + \delta[v']) \\ &= R[v] + \Delta[v'] - \delta[v'] \\ &\geq R[v]. \end{aligned} \tag{4}$$

Algorithm 1 depicts the packet scheduler. The input buffer is organized into a real-time message section and a background traffic section. Each section maintains the front and tail of the corresponding queue. These sections are stored in a circular fashion. The growing of one end of a section can reach the other end of the other section. An incoming real-time message is stored at any end with sufficient space in the real-time message section. If neither ends have sufficient space, background packets at the tail of the background section are dropped until space is sufficient for the real-time message. An incoming background packet is dropped immediately if the input buffer has insufficient space. Otherwise, if the background traffic section has insufficient space at its tail, real-time messages are moved from the head to the tail of the real-time section. An incoming background packet is always stored at the tail of the background traffic section to support an FCFS service. The scheduler scans the real-time messages in the input buffer and processes the message with the smallest $A[v]$ but $t_{current} \geq A[v] - R[v]$. The second condition is to guarantee that a message will not be processed before its latest release time (see lines 4 to 31). In the algorithm, $\bar{A}$ represents the earliest transmission time of all real-time messages in the intermediate queue. The scheduler compares $\bar{A}$ with the current time to determine whether the transmission time of any message has passed. The scheduler transmits all messages whose transmission times have passed (see lines 32 to 36).

```
1  Message Scheduler (node v)
       Input: Packet Input Buffer Q
2      Ā ← ∞
3      while true do
4          t_current ← current time
5          target real message x ← nil
6          target background packet y ← nil
7          for each packet m in Q do
8              if m is a real-time message then
9                  if t_current ≥ A_m[v] − R_m[v] then
10                     if x = nil or A_m[v] < A_x[v] then
11                         x ← m
12                     end
13                 end
14             else
15                 if y = nil then
16                     y ← m
17                 end
18             end
19         end
20         if x ≠ nil then
21             if Ā > A_x[v] then
22                 Ā ← A_x[v]
23             end
24             v′ ← next node for x
25             calculate latest transmission time for next node A_x[v′].
26             move message x into intermediate queue.
27         else
28             if y ≠ nil then
29                 move y into output queue
30             end
31         end
32         t_current ← current time
33         while t_current >= Ā do
34             forward message with transmission time Ā to output interface.
35             update Ā.
36         end
37         T_s ← Ā − t_current
38         if Q is empty then
39             sleep (T_s) or wake up by packet arrival interrupt.
40         end
41     end
```

Algorithm 1: Pseudocode for message scheduler.

Our scheduler belongs to the class of non-preemptive EDF scheduling algorithms. The real-time messages can be considered as jobs of periodic tasks (i.e., real-time message flows) and instructions (lines 20 to 36) can be considered as the non-preemptive execution of the jobs. Since the expected response time R is not required to be equal to the message flow period, we utilize existing processor-demand analysis [2, 13] to perform a schedulability test for the message flows on each node to determine whether the real-time messages can be transmitted on time with the corresponding expected response time. This schedulability test is adopted by the static routing algorithm when deriving message forwarding paths in Section 2.3.

The runtime response time variation, δ, for a particular message consists of two parts. One part is the time to execute at most one iteration of the while loop (lines 4 to 31). The other part is the time to transmit other messages that have earlier transmission times in the intermediate queue than the transmission time of that particular message (lines 32 to 36). Thus, the worst case variation, Δ, is expressed by Eq. 5, where c is the worst case execution time in the first part, $B[v]$ is the buffer size of node v, and $E[v, v']$ is the bandwidth of the link that transmits any real-time messages from node v to node v'. The second part in Eq. 5 represents the time for the data to be transmitted on node v via the slowest link.

$$\Delta[v] = c + \frac{B[v]}{min\{E[v, v']\}}, \quad \textit{for all nodes v' linked to v for real-time messages.} \tag{5}$$

2.3 Routing Algorithm

Two significant differences between the objectives of our routing algorithm and other routing algorithms (see Section 5) are: (1) Our algorithm finds forwarding paths for multiple pairs of source and destination, and (2) it bounds the end-to-end delay for every pair by a predefined value (relative deadline of the flow) instead of minimizing the delay for a particular flow. Thus, we want to increase the predictability of the node behavior so that the message response time on each node is controlled and no real-time messages will be dropped.

Let us first describe the validation algorithm that checks if a set of real-time message flows with their corresponding paths is schedulable. Given a set of flows, a set of corresponding paths (one path for each flow) is schedulable if the following criteria can be met on every node in the network: (1) When a real-time message arrives at node v, the node is able to process it before time $A[v]$ (i.e., latest message response time). In addition, the node is able to transmit the message at a time in the range $(A[v], A[v] + \Delta[v])$. This is to guarantee that the message arrival time at the next node on the path can be modeled by Eq. 1 since this criterion infers $0 \leq \delta[v] \leq \Delta[v]$, which is the assumption for Eq. 1. This criterion is checked for the given message flows and the corresponding paths by the schedulability test described in Section 2.2.

(2) The residual buffer size of every node (i.e., the node buffer size minus the aggregate size of real-time messages that are resident on that node) cannot be negative at any time. Let us first consider the worst case for one message flow m on any node v. The longest time that any message in the flow is queued on this node is $\Delta[v'] + R_m[v] + \Delta[v]$, where v' is the predecessor of node v on the path. Let $\Delta[v'] = 0$ if v is the first node on the path. Thus, the number of messages of this flow on node v in the worst case is $\lceil \frac{\Delta[v'] + R_m[v] + \Delta[v]}{T_m} \rceil$. Then, the aggregate size of all message flows in the worst case must not exceed the buffer size of node v as expressed in Eq. 6. This buffer size limitation has to be validated on every node.

$$\sum_{\textit{all flows } m \textit{ on } v} \lceil \frac{\Delta[v'] + R_m[v] + \Delta[v]}{T_m} \rceil * S_m \leq B[v]\,. \tag{6}$$

With this validation algorithm, our routing algorithm derives the forwarding paths for message flows with the following backtracking procedure. Assuming a schedulable forwarding path set is found for the first i flows, the routing algorithm checks every path candidate for flow $i+1$ until it finds a candidate that will result in a schedulable path set for all first $i+1$ flows according to the validation algorithm. If no such forwarding path is found for flow $i+1$, the algorithm backtracks to flow i and continues the process with the next candidate for flow i. A forwarding path candidate for a message flow is defined as a sequence of nodes that can transmit this flow from its source node to its destination with the expected response time on each node (i.e., $(v_0, R[v_0]), (v_1, R[v_1]), ..., (v_k, R[v_k])$). Section 2.4 describes the algorithm to find path candidates for message flows.

This algorithm always finds a schedulable forwarding path for each message flow if such a path exists since it checks all permutations. The complexity of this backtracking algorithm is exponential with the number of message flows (i.e., the same as an ILP algorithm) but this has no impact on real-time messaging as the calculations are performed offline. Nonetheless, we reduce the actual search time by optimizing the path search order. When checking the path candidates for flow $i+1$, the intuition is to give higher preference to the candidate that results in a larger residual buffer on the path if that candidate is chosen to be the forwarding path for flow $i+1$. The residual buffer size of a path is defined as the minimum residual buffer size of all nodes on the path. Since the residual buffer size of a node determines the size of background traffic that can be kept on the node, this strategy decreases the chance that certain nodes become the bottleneck for the background traffic. If two candidates result in the same residual buffer size, we give higher preference to the candidate with a shorter length. Furthermore, we give higher preference to the candidate with a shorter end-to-end delay if two candidates have the same length. The backtracking algorithm checks candidate paths for flow $i+1$ from highest preference to lowest.

2.4 Forwarding Path Candidate

Our routing algorithm requires that forwarding path candidates for message flow i are known when the algorithm attempts to find the path for that flow. A forwarding path candidate includes the nodes on the path and the expected message response time R on each node. First, we perform a breadth-first-search algorithm on the network graph to find all acyclic paths that connect the source and destination of the message flow. Then, for each path, we need to assign the expected message response time to each node based on the following three considerations.

First, the buffer size restriction described in Eq. 6 must be met on every switch. Second, the overall end-to-end delay in the worst case based on the expected message response time assignment must not exceed the relative deadline of the message flow (as shown in Eq. 3), which can also be expressed as:

$$\sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] + \sum_{j=0}^{k} R[v_j] + \sum_{j=0}^{k} \Delta[v_j] \leq D.$$

$$\text{Thus,} \quad \sum_{j=0}^{k} R[v_j] \leq D - \sum_{j=0}^{k-1} Pr[v_j, v_{j+1}] - \sum_{j=0}^{k} \Delta[v_j]. \tag{7}$$

Third, the assigned expected response time must be long enough on every node v_j on the path so that these messages can be processed with their local relative deadline $R[v_j]$ as required by the validation algorithm. Since the message scheduler is non-preemptive, we use the worst

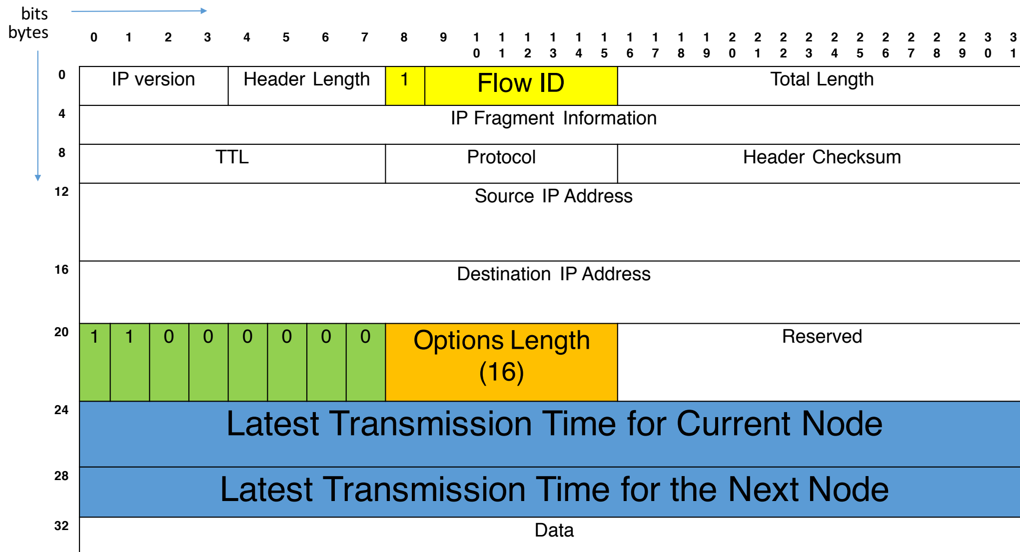

Figure 2 Message Header.

case execution time for the node to process one message (i.e., notation c as described in Section 2.2) as the unit to calculate the expected response time. Thus, $R[v_j] = c * x$, where x is at least 1 and upper bounded by Eq. 6 and 7. The assignment algorithm enumerates all values x in its range and performs processor-demand analysis for EDF scheduling to check if the corresponding response time assignment for the chosen x can be met on the network devices. As a result, multiple forwarding path candidates can be generated for each acyclic path found by the breadth-first-search algorithm.

3 Implementation

To support the three-phases message transmission (see Fig. 1), this section first presents the message structure and the extension to the Linux network stack on end nodes to specify the flow id and release time of real-time messages. It then presents the structure of forwarding tables on network devices and a scheduler implementation on virtual switches based on Open vSwitch [20, 5], a software-defined network (SDN) simulation infrastructure.

3.1 Message Structure and Construction

To support the scheduler in Algorithm 1, real-time messages need to carry the flow id, the latest transmission time for the current node $A[v]$, and the latest transmission time for the next node $A[v_{next}]$. We utilize the *Type of Service (ToS) field (i.e., bits 8–15)* and *Options field* in the standard IP Header to store them as illustrated in Fig. 2. For real-time messages, *bit* 8 is set to 1 and *bits* 9–15 are set to the message flow id. Then, a 16-bytes option (*bytes* 20–35) is used to carry the two time values in milliseconds. The type of the option (*byte* 20) is set to a value that has not been used by other protocols to prevent conflicts. For non real-time traffic, *bit* 8 is set to 0, which is the default value for the ToS field.

When a task running in user mode attempts to transmit a real-time message, it needs to specify the flow id to the network stack of Linux. The network stack software of the

node searches in the forwarding table to calculate the transmission times and determines the network interface for the transmission. Below are the most significant changes we made to Linux to support this functionality.

(1) We added a new field, fid, of type *char* in the kernel data structure *sock* so that the flow id of a socket provided by the task can be stored.

(2) We extended function *sock_setsockopt* of the Linux kernel and added a new option SO_FLOW so that the system call *setsockopt* assigns the message flow id when the task attempts to transmit a real-time message. *sock_setsockopt* keeps the value of the flow id in the fid field of the *sock* structure. The fid field is initialized to 0 when the *sock* structure is created.

(3) When the application transmits the real-time message, the kernel creates instance(s) of the *sk_buff* structure to store the data of the message. We added a new field, fid, in *sk_buff* so that the flow id of the message can be copied and passed down to the network layer. In addition, we added another field, *release_time* of type *ktime_t* in *sk_buff*, to store the current system time as the message release time.

(4) When the network header structure *network_header* in *sk_buff* is constructed in the network layer, we use fid to search in the forwarding table (see Section 3.2) and calculate the latest transmission time for the current node and for the next node. Then, fid and the transmission times are stored in the network header as shown in Fig. 2. *Bit* 8 is set to 1 to indicate real-time messages. If fid is 0, *bit* 8 is set to 0.

(5) We implemented a delay queue, which provides the standard interfaces of the Linux traffic control queue disciplines [11], so that real-time messages can be transmitted at its latest transmission time as required by our routing algorithm. The delay queue contains two components. The first one is a linked-list based FIFO queue to store non real-time packets. For real-time packets, we utilize the *cb* field, the control buffer in *sk_buff*, to implement a linked list-based min-heap. This linked list-based implementation does not have a limit on the number of messages that can be queued in the min-heap, which an array-based implementation of min-heap would have.

(6) When the clock reaches the latest transmission time of a real-time message, the queue discipline moves the value of the latest transmission time for the next node (i.e., *bytes* $28-31$) to *bytes* $24-27$. Then, the message is forwarded to the network interface for transmission. In this way, when the message arrives at the next node, *bytes* $24-27$ denote the latest transmission time for the new node.

3.2 Forwarding Table Structure

We utilize the default forwarding table for background traffic. The default forwarding table contains the mapping between the destination network IDs (i.e., network destination and network mask) and the local interfaces that reach the destinations. For real-time messages, we use the flow IDs to indicate the destinations in the forwarding table. The forwarding table contains the expected message response time ($R[v]$) on the current node v. In addition, it contains the sum of the worst case message transmission variation $\Delta[v]$ on the current node, the propagation delay ($Pr[v, v']$) on the link to the next node v', and the expected message response time ($R[v']$) on the next node. Table 2 depicts the table structure, where *Next Aggregate Delay* is the sum of $\Delta[v]$, $Pr[v, v']$, and $R[v']$. $Interface$ is the interface to forward messages in each flow.

As a result, the time variables of real-time messages required by the scheduler (see Algorithm 1) can be calculated from the data carried in the header and stored in the forwarding table. Table. 3 depicts these relations.

Table 2 Forwarding Table.

fid	Expected Response Time (ms)	Next Aggregate Delay (ms)	Interface
1	7	10	eth0
2	12	16	eth1

Table 3 Message Scheduler Variables.

Variable	Source
$A[v]$	Message header ($bytes$ $24-27$)
$R[v]$	Forwarding table
$A[v']$	$A[v]$ + *Next Aggregate Delay* in forwarding table

3.3 Virtual Switch Implementation

We extend the traffic control [11] and the OpenFlow implementation in Open vSwitch [20, 5] on Linux to implement our message scheduler. Our virtual switch implementation includes three components: (1) An ingress queue based on the traffic control mechanism (i.e., the input queue in our node model). When this queue is enabled, the incoming packets on related interfaces are put into this queue. Meanwhile, the packet arrival events trigger the scheduler to invoke the *dequeue* function if the scheduler is idle. The *dequeue* function finds the packet with the earliest expected response time and sends it to the downstream OpenFlow actions (lines 4 to 19 of Algorithm 1). (2) A new flow table structure (see Section 3.2) is constructed for real-time messages based on the OpenFlow hierarchy and the corresponding forwarding actions to execute (in lines 20 to 31). The scheduler invokes the forwarding actions for every packet. It then forwards real-time messages to the downstream intermediate queue and forwards background traffic to the corresponding interface. (3) A delay queue based on the traffic control mechanism (i.e., the intermediate queue in our node model). This queue has the same structure as the delay queue at the end nodes (see Section 3.1). The scheduler forwards any real-time messages removed from the delay queue with the *dequeue* function for the corresponding interface.

The buffer to store packets on a virtual switch is shared by the ingress queue and the delay queue. When an interface attempts to put a background packet into the ingress queue by invoking the *enqueue* function, we check if the available buffer is sufficient for the packet. If it is not, that background packet is dropped. When an interface attempts to put a real-time message into the ingress queue and the available buffer is insufficient for that message, the *enqueue* function drops background packets that are already in the queue so that the real-time message can be stored. In addition, as one of the objectives of our routing algorithm (see the second criterion of the validation algorithm described in Section 2.3), the algorithm guarantees that the buffer always has enough space to store real-time messages if the forwarding path is schedulable. The size of the buffer is a configurable parameter in our implementation, which is set to different values as part of our experimental assessment.

4 Evaluation

We evaluate our virtual switch on a local cluster. The experiments are conducted on 6 nodes, each of which features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32 GB DRAM and Gigabit Ethernet.

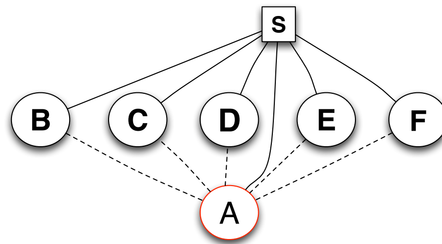

Figure 3 Experiment Network Setup (1).

These nodes are connected via a single network switch in the physical network topology, and we use different numbers of nodes as virtual SDN switch nodes in different experiments. Each node runs a modified version of Linux 2.6.32 and Open vSwitch 2.3.2, which includes the virtual switch implementation. The clocks on these nodes are synchronized with a centralized NTP server when we conduct the experiments. In the first part of this section, we present the experimental results to demonstrate the capabilities of the message scheduler in two aspects: (1) Under intense network congestion, the message scheduler on a single node can meet the hard deadline of every real-time message. (2) When the forwarding paths of real-time messages consist of multiple switches, our message schedulers on these switches can collaborate to guarantee end-to-end deadlines of all real-time messages. In the second part of this section, we present a demonstration for the routing algorithms.

4.1 Single Virtual SDN Switch Result

Background: To demonstrate the effectiveness of the packet scheduler, we measures the deadline miss rate for real-time messages and packet drop rate for background traffic on a single virtual SDN switch with different configurations.

All 6 nodes (marked as A to F in Fig. 3) are connected via a single physical switch (marked as S; physical links are indicated by solid lines). We use node A as the virtual switch. The traffic is transmitted through node A via the virtual links (dashed lines). To support this, a node generates test packets using the IP address of node A as the destination. When a test packet arrives at virtual switch A via the physical links, A uses the flow id carried in the packet header to determine its real destination. Thus, we add an extra column in the forwarding table to indicate the real destination of a message flow. In addition, the ToS field in background packets generated for test purposes is used to indicate their real destination. Virtual switch A fills the real destination of any packet into the IP header of the packet and then forwards it via the physical link to the central switch. In this way, test packets are transmitted to its destination via the virtual switch. This modification is due to the limitation of our experimental environment, which would be unnecessary if the virtual switch were deployed directly onto a physical switch.

We use the network benchmark Sockperf to generate test workloads. A Sockperf client transmits messages as UDP packets to the corresponding Sockperf server via the virtual switch. The client generates a log record to indicate the transmission time of a message while the server generates a log record to indicate the message arrival time. To simplify measurements, the data section for a real-time message contains the flow id and the sequence id of a particular message in that flow. The data section is padded by *0*s so that its total size is 1000 bytes. A packet has been dropped by the virtual switch if no corresponding record exists on the server side after the experiment. We assume that no packet drop occurs in

Table 4 Test Workload in Single Switch Experiment.

fid	Type	Source	Destination	(mps, burst)	Relative Deadline
1	Real Time	B	C	(250, 1)	24ms
2	Real Time	C	D	(200, 1)	24ms
3	Background	D	E	(200, 400)	NA
4	Background	E	F	(200, 400)	NA
5	Background	F	D	(200, 400)	NA

the network stack software of the end nodes or on the physical network links. Our platform meets this assumption since we do not limit the buffer size on end nodes and the physical network is reliable in our cluster.

Table 4 depicts the workload in the first experiment. The workload is controlled by Sockperf parameters $(mps, burst)$. mps indicates the number of frames per second, which affects the frame size. $burst$ defines the number of packets transmitted in each frame by the client. For example, $(250, 1)$ indicates 250 frames per second (i.e., a frame size of $4ms$) with 1 message transmitted per frame. Real-time message flows have fixed parameters to make them strictly periodic. The actual burst of a background flow is a random value uniformly drawn from the interval $[\frac{burst}{2}, burst]$ in each frame. If a real-time message does not arrive at the server (i.e., no corresponding record exists in the server logs), we consider it a deadline miss in this experiment. No re-transmission is performed. In addition, we calculate the end-to-end delay for real-time messages even if they arrive at the server side. If the delay of a message is longer than its deadline, we also consider it as a deadline miss. We set the expected response time on the virtual switch to $20ms$ for both real-time flows, which is the relative deadline of the message flows ($24ms$) minus the aggregate propagation delay and expected response time variations on the path (estimated as $4ms$). The case study in Section 4.3 demonstrates this calculation.

Analysis: Fig. 4 depicts the result. We adjust the buffer size and compare both deadline miss rate and packet drop rate when the message scheduler is turned on and off on the virtual switch. When the message scheduler is turned off, the virtual switch does not differentiate real-time packets from background packets and forwards them to the destination directly. The x-axis in Fig. 4 is the buffer size on the virtual switch. The y-axis is the packet drop rate for background traffic and the deadline miss rate for real-time flows.

The black and green lines depict the deadline miss rate for real-time messages when the scheduler is turned off and on, respectively. With $20ms$ as the expected response time, the message flows are schedulable under all buffer size configurations. Both message flows have a 0% deadline miss rate when the scheduler is on.

▶ **Observation 1.** *The packet scheduler can meet the deadline requirements of all real-time messages.*

The black and blue lines depict the deadline miss rate and packet drop rate when the scheduler is turned off. Since real-time packets and background packets are processed in the same way by the switch, the deadline miss rate and packet drop rate are close for all buffer size configurations. The virtual switch has an increasing tolerance to the burstiness of packets with a larger buffer size. As a result, both rates decrease when the buffer size is increased as depicted by the declining lines.

▶ **Observation 2.** *When the scheduler is off, a larger buffer size can decrease both the deadline miss rate for real-time messages and the packet drop rate for background traffic.*

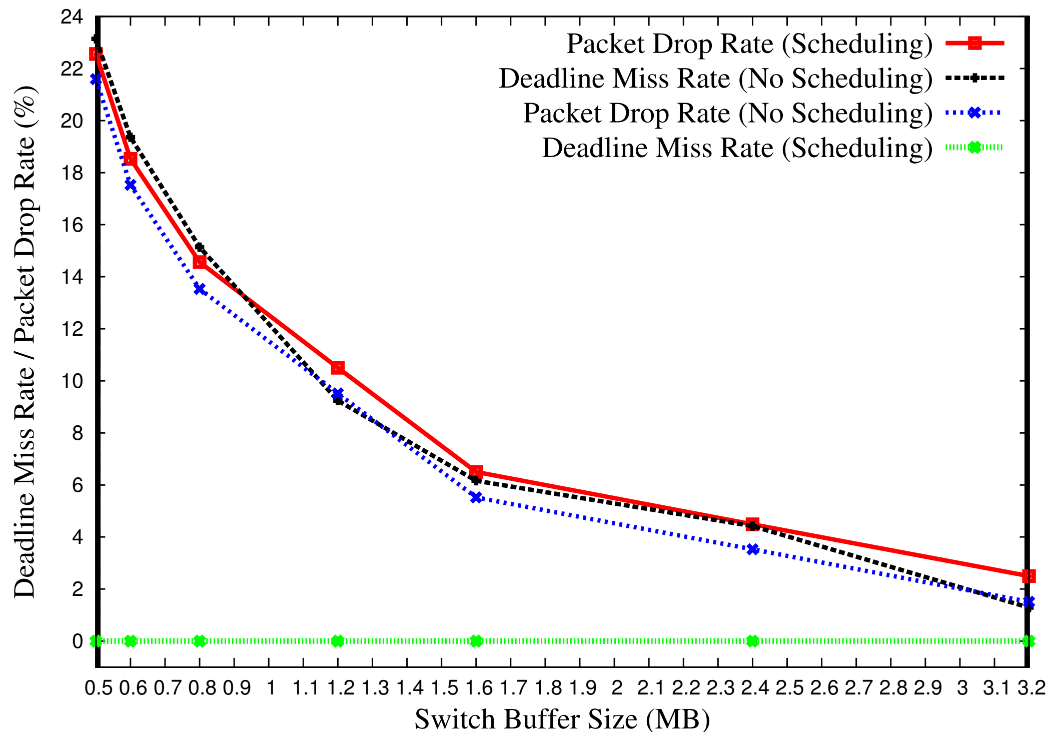

Figure 4 Deadline Miss Rate and Message Drop Rate.

Table 5 Expected Response Time vs Message Drop Rate.

Expected Response Time (ms)	Message Drop Rate (%)
10	10.80
20	10.89
40	11.05
80	11.46
160	11.81
320	12.33

When the scheduler is on, the expected message response time on the virtual switch is $20ms$, which means every real-time message is delayed in the switch buffer for $20ms$ (no delay is added in the end nodes where the real-time messages are released and received). Due to this delay, the available buffer size for background traffic shrinks. As a result, the packet drop rate for background traffic increases slightly, e.g., from 17.53% (blue line) to 18.53% (red line) for a buffer size of $0.6MB$. Table 5 shows an overall trend of an increasing drop rate for background traffic when the expected response time for real-time messages is increased from $10ms$ to $320ms$ for a buffer size of $1.2MB$. The results indicate that the longer real-time messages are delayed in the switch buffer, the smaller the available buffer is for background traffic. As a result, the packet drop rate increases when the delay time for real-time messages is increased.

▶ **Observation 3.** *Our scheduler increases the packet drop rate for background traffic.*

The measurement of end-to-end delays for real-time messages indicates that the message scheduler does not introduce a significant variation to the message transmission time (Δ is

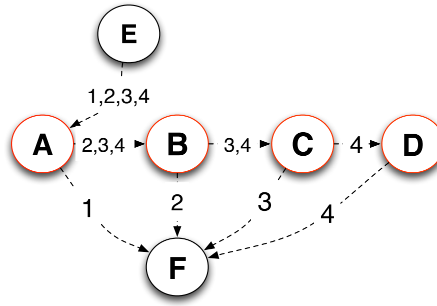

Figure 5 Experiment Network Setup (2).

small). The end-to-end delay includes the latency due to transmission from the source node to the virtual switch, the time the message spent on the virtual switch, and the transmission latency from the virtual switch to the destination node. When we set the buffer size to $1.2MB$ and the expected response time to $10ms$, the shortest end-to-end delay is $10.9ms$ and the longest is $12.1ms$ (the median value is $11.6ms$), of which $10ms$ are due to the software-induced delay on the virtual switch. Since messages are transmitted via a physical switch (and not via direct links in our system model), the physical switch also contributes to the delay variation.

4.2 Multiple Virtual Switches Result

Background: In the second experiment, we construct a virtual switch chain to transmit real-time messages. Nodes $A - D$ are configured as virtual switches with a buffer size of $1.2MB$. Node E uses a Sockperf client to transmit four message flows to the Sockperf server on node F. Fig. 5 depicts the forwarding path for each message flow. The periods of message flows are $1ms$. The expected response time is $5ms$ and the next aggregate delay is $7ms$ for each flow on these virtual switches. This setup suggests that the sum of the worst case response time variation, Δ, and the propagation delay, Pr, on the link between two virtual switches (i.e., two physical links connected via the central physical switch) is $2ms$. We measure the end-to-end delay for each message.

Our experimental results show that messages in all four flows experience a similar end-to-end delay variation ($[0.8ms, 1.8ms]$). The value of next aggregate delay includes the worst case response time variance on the previous node (as described in Eq. 3.2). The scheduler only considers a real-time message once the current time is greater or equal to the worst case response time (see line 9 in Algorithm 1). As a result, only the response time variation on the last virtual switch (plus the propagation delay variation on the physical link from the last virtual switch to the receiver in our experiment) contributes to the measured variation of the end-to-end delays even though the forwarding path has multiple virtual switches. This indicates the effectiveness of the message scheduler in terms of variance control.

▶ **Observation 4.** *Real-time messages do not accumulate response time variation (jitter) when forwarded by a chain of virtual switches.*

4.3 Routing Algorithm Demonstration

We demonstrate our routing algorithm to find forwarding paths for real-time message flows. Three message flows with attributes indicated in Table 6 are considered. These flows can

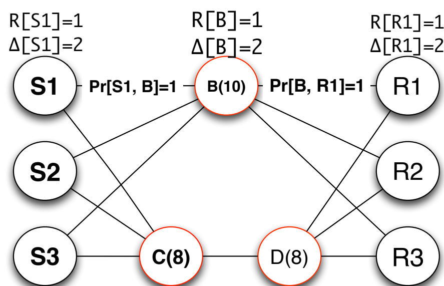

Figure 6 Routing Algorithm Demonstration.

Table 6 Real-time Message Flows in Demonstration.

fid	Source	Destination	Period (T)	Relative Deadline (D)	Message Size (S)
1	S1	R1	12	11	1
2	S2	R2	1	15	1
3	S3	R3	12	12	9

be either forwarded via intermediate switch B or switches C and D as depicted in Fig. 6, where the numbers in the switch circles are the buffer size of that switch. For simplicity, we consider that it takes $1ms$ for the message scheduler on every switch to process one message (i.e., $c = 1ms$ in Eq. 5). We consider a response time variation of $2ms$ on every node or switch and a propagation delay of $1ms$ ($Pr = 1ms$) on every link (i.e., $\Delta = 2ms$, $Pr = 1ms$).

We use the same notation as described in Section 2.3 to express forwarding path candidates. For example, candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 means that a real-time message of flow 1 is forwarded via nodes $S1$, B, and $R1$ with expected message response times of $1ms$, $1ms$, and $1ms$ on each node. We determine the schedulable routing paths for this setup in the following steps.

(1) Consider forwarding path candidate ((S1, 1), (B, 1), (R1, 1)) for flow 1 (see 1st row in of Table 7). The worst case end-to-end delay of this path candidate is $R[S1] + \Delta[S1] + Pr[S1, B] + R[B] + \Delta[B] + Pr[B, R1] + R[R1] + \Delta[R1] = 11ms$. Thus, this is the only forwarding path candidate for flow 1 according to the deadline constraint of Eq. 7. If the expected response time on any node of the path were increased, the end-to-end delay would exceed the relative deadline of flow 1, which is $11ms$.

The size of messages in flow 1 is 1. The maximum number of messages in flow 1 that could be on switch B at any time is $\lceil \frac{\Delta[S1]+R[B]+\Delta[B]}{T_1} \rceil = 1$. As a result, the residual buffer size of node B becomes 9, which is the buffer size of node B (10) minus the maximum buffer that could be occupied by messages in flow 1.

(2) The forwarding path candidates for flow 2 are shown with ID 2-6 in Table 7, where columns B, C, and D depict the residual buffer size on the corresponding nodes if flow 2 is forwarded. Candidates 2, 3, and 4 are considered first since the residual buffer size on these paths is 4 (i.e., $B - \lceil \frac{\Delta[S2]+R[B]+\Delta[B]}{T_2} \rceil * S_2 = 9 - 5 = 4$) if flow 2 is scheduled, which is larger than the candidates 5 and 6. However, switch B cannot schedule both flow 1 and flow 2, since the utilization of flow 2 is $\frac{c}{T_2} = 100\%$. Other path candidates for flow 2 with $R[B] = 1ms$, which are not shown in Table 7, are rejected for the same reason.

(3) Candidate 5 is considered next for flow 2 since it is a shorter path compared to candidate 6 even though they have the same residual buffer size on their paths. However, if

Table 7 Forwarding Path Candidates.

ID	Flow	Forwarding Path Candidate	End-to-end Delay	Residual Buffer[1)]			Result
				B	C	D	
1	1	(S1, 1), (B, 1), (R1, 1)	11	9	8	8	✔
2	2	(S2, 1), (B, 1), (R2, 1)	11	4	8	8	✕
3	2	(S2, 2), (B, 1), (R2, 1)	12	4	8	8	✕
4	2	(S2, 1), (B, 1), (R2, 2)	12	4	8	8	✕
5	2	(S2, 1), (B, 2), (R2, 1)	12	3	8	8	✕
6	2	(S2, 1), (C, 1), (D, 1), (R2, 1)	15	9	3	3	✔
7	3	(S3, 1), (B, 1), (R3, 1)	11	0	3	3	✔
8	3	(S3, 1), (B, 2), (R3, 1)	12	0	3	3	✕
9	3	(S3, 2), (B, 1), (R3, 1)	12	0	3	3	✕
10	3	(S3, 1), (B, 1), (R3, 2)	12	0	3	3	✕

1) After the corresponding flow is scheduled on the path.

flow 2 is transmitted via candidate 5, the residual buffer size of node B becomes 3. In this case, flow 3 has no forwarding path candidate since flow 3 requires that the buffer size of any node on the path is at least 9, the size of the message in flow 3, since the other forwarding path (via switches C and D) only has a residual buffer size of 8. Other path candidates for flow 2 with $R[B] \geq 2ms$, which are not shown in Table 7, are rejected for the same reason. Thus, the algorithm has to consider candidate 6 in Table 7, which is chosen.

(4) Candidates 7–10 for flow 3 meet the buffer size requirement and the relative deadline of flow 3. Candidate 7 has a higher preference since it has a shorter end-to-end delay. In addition, node B can schedule both flow 1 and 3, both with an expected response times of $1ms$. In summary, we found paths for all three flows.

5 Related Work

Our packet scheduler adopts per-node traffic control to guarantee transmission deadlines similar to other switched real-time Ethernet mechanisms, e.g., rate-controlled service disciplines (RCS) [8, 33], token-bucket traffic shaping [17], and EDF scheduling [9, 34]. Unlike these mechanisms, our scheduler considers multiple constraints imposed by network devices, namely link speed, computation speed, and buffer capacity. Table 8 details the comparison of the assumptions of these mechanisms. $defined$ in the table indicates that the corresponding mechanism considers the restriction in its model. Our work does not assume infinite buffer capacity and computation capacity. Our assumptions are more realistic on commodity switches connected by modern high speed links.

Past work has proposed different mechanisms to establish communication channels that recognize real-time requirements of packet flows at run time [7, 9, 26, 17]. In contrast, our work focuses on forwarding path planning (via a static routing algorithm) and forwarding policy enforcement (via packet scheduling). Other mechanisms guarantee soft deadlines of real-time packets while providing high throughput to other traffic [28], or adopt congestion avoidance algorithms when network contention occurs (e.g., buffer occupancy exceeds a certain threshold) [29]. Our work guarantees hard deadlines by dropping only non-realtime packets upon network congestion.

Our schedulability model does not depend on statistics as metrics of the network, e.g., bandwidth. Such metrics are dependent on multiple primitive factors of the switch: the

Table 8 Assumptions Comparison for Real-time Packet Schedulers.

Mechanism	Buffer Capacity	Computation Capability	Link Speed
RCS [8, 33]	infinite	infinite	defined
EDF [9, 34]	infinite	infinite	defined
Traffic shaping [17]	defined	infinite	defined
D^3 [29], D^2TCP [28]	defined	infinite	defined
Our work	defined	defined	defined

packet scheduling algorithm, buffer size, computing capability, and the state of other flows on the switch [10]. Instead, we have specifically considered these factors in two ways: (1) We formalize the dynamics of the network delay by introducing response time variations (δ) in our analysis and we aggressively control the variation by the message scheduler. (2) Our routing algorithm considers multiple constraints on switches and links when finding forwarding paths.

In contrast to TDMA-based real-time Ethernet channel implementations [4, 15], which rely on custom hardware to respond to control data frames, our scheduler is designed and implemented on Linux-compatible virtual switches and can be deployed on modern commodity SDN-compatible switches, which is more suitable for wide-area deployment.

Past work has studied routing algorithms to find packet forwarding paths with QoS support in different situations. This includes Dijkstra's algorithm to find the single-source path with shortest end-to-end delay under the assumption that per switch delay is known as priori [12], Suurballe's algorithm to find multiple disjoint paths with minimal total end-to-end delay [27], and methods to find multiple disjoint paths with the minimized delay of the shortest path [30]. Others have studied the routings to find an optimal forwarding path for one particular message flow under certain network assumptions [12]. Past work has also proposed to transmit messages over multiple paths simultaneously while the total time of the flow is constrained under the assumption that actual bandwidths are known [24]. Network Calculus has also been adopted to derive the forwarding path for a particular flow under the additional assumptions that no cross-over traffic exists or the pattern of cross-over traffic is known as priori [3]. Our work differs as follows from these prior work. It derives forwarding paths for multiple message flows to guarantee the hard deadline of every message. Our analysis depends on the link speed matrix, E, as described in Section 2. However, when the implementation is deployed on physical switches, E can be quantified by the medium speed, which is independent of the network traffic and scheduling algorithms. In addition, our routing algorithm belongs to the class of constraint-based path selection algorithms, where multiple constraints are considered on the transmission links [16]. We extend that by adding the constraint of switch buffers and formalizing the cost and evaluation functions for switch buffers and message scheduling, which are essential when applying any constraint-based path selection algorithm.

To find forwarding paths for multiple packets, past work has proposed to minimize the average packet delay [12]. However, the objective of real-time message transmission is to meet the deadline of each message flow (i.e., not to minimize the average delay). Our routing algorithm considers the hard deadline of every real-time flow when assigning a forwarding path.

6 Conclusion

We have presented a routing algorithm to determine forwarding paths for real-time message flows in a distributed computing environment and a scheduler to actively enforce a message forwarding policy on network devices. Our routing algorithm considers both the deadlines and the network resource demands of real-time messages. As a result, no real-time messages can be dropped due to network contention when handled by our message scheduler and no real-time messages miss their deadlines. We have implemented the scheduler on virtual switches and conducted experiments on a local cluster to prove the effectiveness of the scheduler. Our experimental results showed that deadline misses of real-time messages dropped to 0 when the message scheduler was turned on.

Future work includes porting the message scheduler implementation to physical network devices (e.g., physical switches). Modern SDN-compatible switches support customized packet forwarding protocols (e.g., OpenFlow), which could be utilized to run our virtual switch implementation based on Open vSwitch. Since physical switches have hardware that is dedicated to packet processing, we expect a better performance than the virtual switches. In addition, in the case where a set of real-time message flows cannot be scheduled on a network environment due to hardware limitations, we plan to extend our routing algorithm to produce suggestive information, e.g., the required increase in buffer size of a switch before the set of flows becomes schedulable. Furthermore, we plan to extend the routing algorithm to find disjoint forwarding paths for real-time messages to handle network device failures.

References

1. Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.
2. Karsten Albers and Frank Slomka. An event stream driven approximation for the analysis of real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 187–195. IEEE, 2004.
3. Anne Bouillard, Bruno Gaujal, Sébastien Lagrange, and Éric Thierry. Optimal routing for end-to-end guarantees using network calculus. *Performance Evaluation*, 65(11):883–906, 2008.
4. Gonzalo Carvajal, Luis Araneda, Alejandro Wolf, Miguel Figueroa, and Sebastian Fischmeister. Integrating dynamic-tdma communication channels into cots ethernet networks. *IEEE Transactions on Industrial Informatics*, 12(5):1806–1816, 2016.
5. Martin Casado, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Rethinking packet forwarding hardware. In *HotNets*, pages 1–6. Citeseer, 2008.
6. Catalyst Switch Architecture. URL: `http://www.cisco.com/networkers/nw03/presos/docs/RST-2011.pdf`.
7. Domenico Ferrari and Dinesh C. Verma. A scheme for real-time channel establishment in wide-area networks. *Selected Areas in Communications, IEEE Journal on*, 8(3):368–379, 1990.
8. Leonidas Georgiadis, Roch Guérin, Vinod Peris, and Kumar N. Sivarajan. Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking (TON)*, 4(4):482–501, 1996.
9. Hoai Hoang, Magnus Jonsson, Anders Kallerdahl, and Ulrik Hagström. Switched real-time ethernet with earliest deadline first scheduling-protocols and traffic handling. *Parallel and Distributed Computing Practices*, 5(1):105–115, 2002.

10 Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125. IEEE, 2008.

11 Bert Hubert, Thomas Graf, Greg Maxwell, Remco van Mook, Martijn van Oosterhout, P. Schroeder, Jasper Spaans, and Pedro Larroy. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, page 213, 2002.

12 Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. In *SIGCOMM'04*, pages 145–158. ACM, 2004.

13 Kevin Jeffay, Donald F Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

14 D. D. Kandhlur, Kang G. Shin, and Domenico Ferrari. Real-time communication in multi-hop networks. *Parallel and Distributed Systems, IEEE Transactions on*, 5(10):1044–1056, 1994.

15 Hermann Kopetz and Günter Grünsteidl. TTP-A time-triggered protocol for fault-tolerant real-time systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers, The Twenty-Third International Symposium on*, pages 524–533. IEEE, 1993.

16 Fernando Kuipers, Piet Van Mieghem, Turgay Korkmaz, and Marwan Krunz. An overview of constraint-based path selection algorithms for QoS routing. *IEEE Communications Magazine, 40 (12)*, 2002.

17 Jork Loeser and Hermann Haertig. Low-latency hard real-time communication over switched ethernet. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 13–22. IEEE, 2004.

18 David L. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

19 Seyedbehzad Nabavi, Jianhua Zhang, and Aranya Chakrabortty. Distributed optimization algorithms for wide-area oscillation monitoring in power systems using interregional pmu-pdc architectures. *Smart Grid, IEEE Transactions on*, 6(5):2529–2538, 2015.

20 Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

21 T. Qian, F. Mueller, and Y. Xin. A real-time distributed hash table. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

22 Tao Qian, Aranya Chakrabortty, Frank Mueller, and Yufeng Xin. A real-time distributed storage system for multi-resolution virtual synchrophasor. In *Power & Energy Society General Meeting*. IEEE, 2014.

23 Tao Qian, Frank Mueller, and Yufeng Xin. Hybrid EDF Packet Scheduling for Real-Time Distributed Systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 37–46, July 2015.

24 Nageswara SV Rao and Stephen G Batsell. QoS routing via multiple paths using bandwidth reservation. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 11–18. IEEE, 1998.

25 Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60. IEEE, 2007.

26 Rui Santos, Moris Behnam, Thomas Nolte, Paulo Pedreiras, and Luís Almeida. Multi-level hierarchical scheduling in ethernet switches. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 185–194. ACM, 2011.

27 John W. Suurballe and Robert Endre Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

28 Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

29 Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

30 Dahai Xu, Yang Chen, Yizhi Xiong, Chunming Qiao, and Xin He. On finding disjoint paths in single and dual link cost networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

31 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.

32 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

33 Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *J. High Speed Networks*, 3(4):389–412, 1994.

34 Kai Zhu, Yan Zhuang, and Yannis Viniotis. Achieving end-to-end delay bounds by edf scheduling without traffic shaping. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1493–1501. IEEE, 2001.