A Failure Recovery Protocol for Software-Defined Real-Time Networks

Tao Qian, Frank Mueller, *Fellow, IEEE*

***Abstract*—In a distributed computing environment, real-time tasks communicate via a network infrastructure whose stability significantly impacts timing predictability. Network stability includes two aspects. First, the network has to guarantee the deadline requirements of real-time message transmissions in the absence of network failures. Second, the network needs to support dynamic recovery when network failures occur. This work generalizes previous static routing approaches, which address the first aspect of the network stability, by developing a dynamic failure recovery policy and a protocol to address the second aspect of the network stability. We derive new real-time forwarding paths without compromising the capability of network devices to guarantee deadlines of concurrent real-time transmissions. We implement this mechanism on a network simulation platform and evaluate it on real hardware in a local cluster to demonstrate its feasibility and effectiveness. Experiments confirm the ability to bound recovery delays based on the network parameters.**

***Index Terms*—Distributed Real-Time Systems, Software-Defined Networking, Bounded Network Failure Recovery**

I. Introduction

A common feature of real-time tasks running in distributed systems is that they require their message flows to be transmitted in a timely manner so that these tasks can be finished before their deadlines. A message flow consists of periodic messages from one real-time task to another. In distributed real-time systems, the set of such messages flows has to be known a priori so that reservations ensure end-to-end transmission times compliant with deadline constraints. In order to realize such a transmission, several scheduling mechanisms have been proposed to control how the underlying network transmits messages on network devices, e.g., rate-controlled service disciplines [4], [16] and earliest-deadline-first (EDF) packet scheduling [5], [17]. However, network dynamics increase the complexity of designing a reliable real-time network. Consider an Ethernet link or switch failure. Message flows that originally utilized the failed hardware can no longer be transmitted to their destinations. This results in deadline misses for the corresponding real-time tasks unless a failure handling policy is adopted, such as: (1) Multiple disjoint forwarding paths can be utilized to transmit multiple copies of every real-time message of a flow to avoid deadline misses caused by a single network failure [10]. As long as at least one path is error-free, at least one message arrives at the destination within its transmission deadline. (2) The system can start a dynamic routing procedure to promptly establish new real-time transmission paths for affected message flows [3], [11], [5]. The first approach has a lower resource utilization rate since it requires one to reserve network resources on all forwarding paths for the same real-time message flow. The second approach has a longer transmission delay since it requires the system to go through a recovery phase to derive new forwarding paths before the messages can be re-transmitted.

The first approach that transmits the same message flow via multiple disjoint paths is necessary for real-time tasks with short deadlines. In this case, the time of reserving new real-time paths for these tasks could already exceed their deadlines. However, in order to handle potential network failures in the future, a dynamic routing algorithm can be adopted. This way, new real-time paths can be established to replace the failed ones.

Upon experiencing a network failure, a failure detection mechanism first needs to recognize the failure and identify affected real-time message flows. Then, a recovery mechanism triggers a dynamic routing algorithm to derive an alternate path for every affected flow. During the routing phase, packets carrying the routing request and the transmission demand of the message flow (e.g., message period and size) are sent. A network switch, upon receiving a routing packet, needs to perform an acceptance test to determine if this message flow can be transmitted via this switch considering its resource limitation (e.g., buffer size, compute capability, and link speed). The routing algorithm needs to reserve network resources on a path along switches for the message flow, and it needs to derive the new path in a timely manner if transmission along the original path failed.

The failure recovery mechanism needs to derive new paths for multiple message flows, which can result in resource conflicts on the switches. Fig. 1 demonstrates one potential resource conflict situation. Message flow-1 was transmitted on the path A→B→E and flow-2 via E→B→A before switch B failed. Once B fails, the recovery mechanism causes the source nodes of both flows to send out routing packets. The new paths would be A→C→D→E for flow-1 and E→D→C→A for flow-2 if both switches C and D had enough resources. Now consider the situation when neither C nor D have enough resources to forward both flows. When the routing packet for flow-1 has reserved the resource on switch C while the routing packet for flow-2 has reserved the resource on switch D, resource conflicts have occurred on switches C and D between the two flows. An effective conflict resolution is required to

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

T. Qian and F. Mueller were with the Department of Computer Science, North Carolina State University, Raleigh, NC, 27695-8206 USA e-mail: mueller@cs.ncsu.edu.

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue.

continue the failure recovery process. Assuming flow-1 is more critical than flow-2, an ideal conflict resolution should only reserve resources on path A→C→D→E for flow-1 instead of reserving resources for flow-2 or none of the two flows.

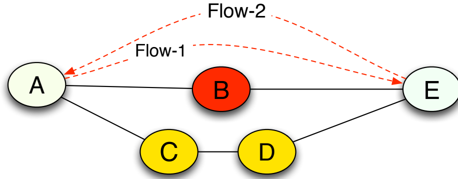


Fig. 1: Resource Conflict Example

Contributions: We propose a hybrid failure handling policy to address these problems with the objective to provide robust, fully distributed resilience to real-time flow. The basic idea is as follows. We divide real-time message flows into two classes according to their urgency, also suitable for mixed criticality, and adopt different recovery policies for them when a network failure occurs. For the real-time message flows in the first class with short deadlines (Class-A), we statically reserve multiple disjoint paths for each flow. For the real-time message flows in the second class with long deadlines (Class-B), we reserve a single path for each flow. When a network failure occurs, the routing requests of Class-A flows initially have a lower priority for deriving new paths than the requests of Class-B flows. The intention is to derive new paths for Class-B flows with a larger success rate and a guaranteed upper bound for the recovery time if the network resources allow, since their tasks are blocked waiting for a new path to re-transmit their messages whereas Class-A messages can use their alternate path. In addition, we adopt a timer-based algorithm to reserve resources distributed on different switches and resolve resource conflicts. This paper focuses on a dynamic failure recovery mechanism, which derives new forwarding paths for real-time message flows.

We obtained a copy of a real-time packet scheduler [9] that operates within software-defined network routers to establish real-time flow guarantees. We develop a failure recovery mechanism in this environment that processes routing packets while still transmitting regular real-time message flows such that their timing requirements are met. We adopt a server dedicated to processing routing packets and transmitting background traffic (i.e., non-realtime packets) when no routing packets are pending. Similar to the Total Bandwidth Algorithm [12], we assign a constant fraction of the network resources to this dedicated server so that the transmission of regular real-time packets is not affected. This is detailed in Section II.

II. Design

This section first presents the background and then details the design of a dedicated server for failure recovery of static routing algorithm and packet scheduling. After that, it details the failure recovery mechanism and the timer-based conflict resolution algorithm. This section is concludes with the timing analysis for failure recovery.

A. Background

We consider three types of packets transmitted in distributed systems. (1) Packets transmitted by real-time tasks: These packets can be classified as message flows. A message flow consists of all the messages periodically released by the same real-time task. Thus, these messages have the same source and destination, relative transmission deadline, and are transmitted via the same forwarding path in the underlying network in the absence of network failures. Our model requires that the set of real-time flows is known a priori. We use the term *real-time packets* or *real-time messages* interchangeably to refer to these packets. (2) *Routing packets*: When a network failure occurs, routing packets are injected into the network to derive new forwarding paths for affected real-time flows. Both real-time packets and routing packets carry their relative deadlines. The packet scheduler on network devices must schedule these packets accordingly in order to avoid deadline misses. (3) Packets transmitted by tasks with no deadline requirements (i.e., *background packets* or *background traffic*): The packet scheduler can drop background traffic when resource contention occurs.

Without loss of generality, we use the term $node$ to refer to either a compute node that executes distributed tasks or a network device that transmits the packets. We require that clock times on nodes are synchronized by either the support of hardware (e.g., global positioning system (GPS) of phasor measurement units in the power grid) or a clock synchronization protocol (e.g., the Network Time Protocol [8] running at system startup time as well as periodically as a real-time task).

Given a set of real-time message flows and the configuration of the underlying network (i.e., the network topology and buffer capacity of each node), past work has proposed a static routing algorithm to derive the forwarding paths for these flows and a packet scheduler to deliver these flows by their local deadlines [9]. The main idea is that the end-to-end transmission delay of a real-time packet can be accurately controlled by processing the packet at the designated time on every node along its forwarding path. Fig. 2 depicts the timeline for a real-time packet arriving at time $t[v]$ on node v, a node on the forwarding path. The static routing algorithm derives the local relative deadline $R[v]$ offline and calculates the latest transmission time $A[v]$ based on the local deadlines of the packet on all nodes along the path.

After arrival, the real-time packet is considered not ready to be processed by the packet scheduler until time $A[v] - R[v]$, when the packet is marked ready with a local deadline of $R[v]$. The packet scheduler employs an EDF algorithm to process all ready real-time packets. This involves searching the node routing table to determine the forwarding policy for the packet (e.g., the output link for the packet). Since the static routing algorithm guarantees that all the real-time packets can be processed before their local deadlines, the packet scheduler can move this packet into the output queue at time $A[v]$. The packet is eventually sent onto the output link at time $A[v] + \delta[v]$, where $\delta[v]$ is the queuing time for the packet in the output buffer. With this algorithm, the scheduler can control the duration for which any real-time packet can stay on a node, thus avoiding resource contention between real-time packets.

The packet scheduler provides a best-effort delivery service to background traffic in two aspects. First, the scheduler only

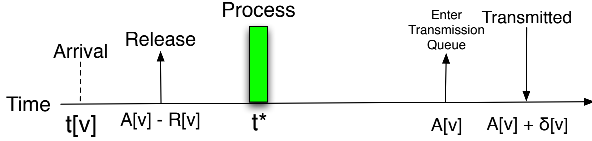


Fig. 2: Real-time Packet Phases on Node

processes background packets when no real-time packets are ready to be processed. Second, when resource contention occurs due to a burst of background traffic, the packet scheduler drops only background packets.

The packet scheduler handles routing packets the same as background packets. Since routing packets are transmitted only when the failure recovery mechanism is triggered due to network failures, they could potentially compete with real-time packets for network resources. However, due to the timing requirements of routing packets, they cannot be dropped arbitrarily like background packets. In Section II-B, we present the extension to the packet scheduler to accommodate routing packets.

B. Packet Scheduling Extension

We utilize a dedicate server, the "routing packets server" (RPS), on each node to process routing packets for dynamic failure recovery. This server can be represented by $RPS(\alpha, \beta)$, where α is the maximal portion of the buffer that can be used to store routing packets on each node and β is the maximal portion of the compute capability that the node uses to process routing packets.

Building on a previous static routing algorithm [9], which derives the forwarding paths for real-time flows, we accommodate the resource demands of server $RPS(\alpha, \beta)$. First, the aggregate size of real-time messages that are resident on any node v cannot exceed $(1-\alpha)B[v]$ at any time, where $B[v]$ is the total buffer size of node v. Second, when processor-demand analysis for EDF scheduling for real-time flows is performed, the total processor utilization of real-time flows cannot exceed $(1-\beta)$.

Furthermore, the following rule determines which packet is dropped when buffer contention occurs on a node. (1) When the arriving packet is a background packet, drop it directly. Background packets have the lowest priority compared to real-time packets and routing packets. The node processes background packets in a FCFS order. (2) When the arriving packet is either a real-time packet or a routing packet, drop background packets to meet the buffer requirement of the arriving packet first. If the buffer is still insufficient, drop routing packets. This happens when the aggregate size of routing packets exceeds the upper memory bound $\alpha B[v]$. We assign different priorities to routing packets. The scheduler drops the routing packets from lowest priority to highest priority until the buffer size is sufficient for the incoming real-time or routing packet. Section II-C presents the prioritization rule for routing packets.

These policies and the packet dropping rule guarantee that real-time packets can neither be dropped nor miss their deadlines once their forwarding paths have been derived offline, since the aggregate size of routing packets does not exceed $\alpha B[v]$ and the node does not utilize more than a proportion of the compute capability β to process routing packets.

1 **Message Scheduler Task** *(node v)*
  Inputs: Packet Input Queue Q, process quota rpq, next transmission time $\bar{A}$
2 $t \leftarrow$ current time
3 target real time packet $x \leftarrow$ *nil*
4 target background packet $y \leftarrow$ *nil*
5 target routing packet $z \leftarrow$ *nil*
6 **for** *each packet m in Q* **do**
7   **if** *m is a real-time message* **then**
8   **if** $t \geq A_m[v] - R_m[v]$ *and (x = nil or $A_m[v] < A_x[v]$)* **then**
9       $x \leftarrow m$
10   **end**
11   **else if** *m is a routing packet* **then**
12   **if** *m is not pending and (z = nil or $m.Priority > z.Priority$)* **then**
13       $z \leftarrow m$
14   **end**
15   **else if** y = *nil* **then**
16     $y \leftarrow m$
17   **end**
18 **end**
19 $flag \leftarrow false$
20 **if** $x \neq$ *nil* **then**
21   process and move x into intermediate queue
22   $flag \leftarrow true$
23 **end**
24 forward message with transmission time $\bar{A}$ to output interface.
25 $rpq \leftarrow rpq$ + (current time - t) * β
26 **if** $x = nil$ *and* $z \neq nil$ *and* $rpq \geq T_{rps}$ **then**
27   RPS: process routing packet z with worst case execution time T_{rps}
28   $rpq = rpq$ - processing time
29   $flag \leftarrow true$
30 **end**
31 **if** $y \neq nil$ *and* $flag = false$ **then**
32   move y into output interface
33 **end**
34 update local resource table driven by timers
35 $rpq \leftarrow rpq$ + (current time - t) * β
36 **if** $z = nil$ *and* $rpq > T_{rps}$ **then**
37   $rpq \leftarrow T_{rps}$
38 **end**

Algorithm 1: Pseudocode for extended message scheduler

Algorithm 1 depicts the packet scheduler extended with the $RPS(\alpha, \beta)$ server. A node executes this algorithm in an infinite loop. Algorithm 1 uses variable rpq to store the remaining CPU quota available for processing routing packets. This quota is accumulated in a ratio of β relative to the time the scheduler has spent on processing packets. A routing packet can only be processed if the quota rpq exceeds the worst case execution time of the RPS task (i.e., T_{rps}, shown in line 26). In Algorithm 1, lines 8 - 10 choose the real-time packet with shortest deadline among all ready real-time packets (i.e., packets in the queue whose release time $A_m[v] - R_m[v]$ has passed). Lines 11 - 14 choose the ready routing packet with the highest priority. Routing packets can be marked as temporarily pending to reduce resource conflicts, which will be discussed in Section II-D. Lines 15 - 16 choose a background packet in a FCFS order. After this, the scheduler processes the real-

time packet x, sends it to the intermediate queue (lines 20 - 23) and moves the real-time packets from the intermediate queue to the output interface at the latest possible time (i.e., when there is just enough time left to accommodate their worst case transmission time). Then, the scheduler invokes RPS to process the routing packet z if the quota is sufficient, i.e., $rpq \geq T_{rps}$ (lines 26 - 30). The scheduler only forwards a background packet y if no real-time or routing packets have been processed in the current loop cycle (lines 31 - 33).

Lines 36 - 38 in Algorithm 1 reset rpq to T_{rps} if no routing packet is currently in the input queue. This is to prevent a delay real-time packet processing when rpq is accumulated to a large amount and a burst of routing packets arrives at the node just at that point in time. Thus, this algorithm enforces that the utilization of RPS never exceeds β.

C. Routing Packets Scheduling

In our system, network failures are detected at the destination nodes of real-time flows. Since the static routing algorithm can derive the expected end-to-end transmission time of a message in any real-time flow, the flow destination can determine whether any of its messages is missing by utilizing synchronized clocks. When this happens, the destination node sends a routing packet into the network to derive a new path for that flow. In addition to the destination and source nodes of that flow, routing packets carry information on the criticality class and the relative deadline of that real-time flow. Two criticality classes are supported by the routing packet server RPS. Since we assume that multiple copies of the same message of a Class-A flow have been transmitted via multiple disjoint paths and at least one of them arrives at the destination within its expected transmission time, RPS assigns a lower priority to the routing packets for Class-A flows than for for Class-B flows. After all, the latter tasks are blocked waiting for a new path to be discovered before they can transmit the messages. Routing packets for flows of the same class are processed in an EDF order (according to the relative deadlines of the flows). Routing packets for the same flow are processed in a FCFS order.

Furthermore, routing packets carry the period of the corresponding real-time flow and the size of a message in that flow. RPS on a node uses this information to perform an acceptance test, which checks if the aggregate size of all real-time messages does not exceed the $(1-\alpha)B[v]$ bound and the utilization does not exceed $(1-\beta)$, as presented in Section II-B, for a real-time flow transmitted via this node. We use the predicate $acceptable(v, f)$ to represent the acceptance test, which returns $true$ iff flow f passes the test on node v. If the acceptance test is passed, this node becomes one of the node candidates for the real-time flow. The goal of the routing algorithm is first to find a new path that consists of node candidates between source and destination of the flow. The routing algorithm then reserves the resources on a single new path for that flow.

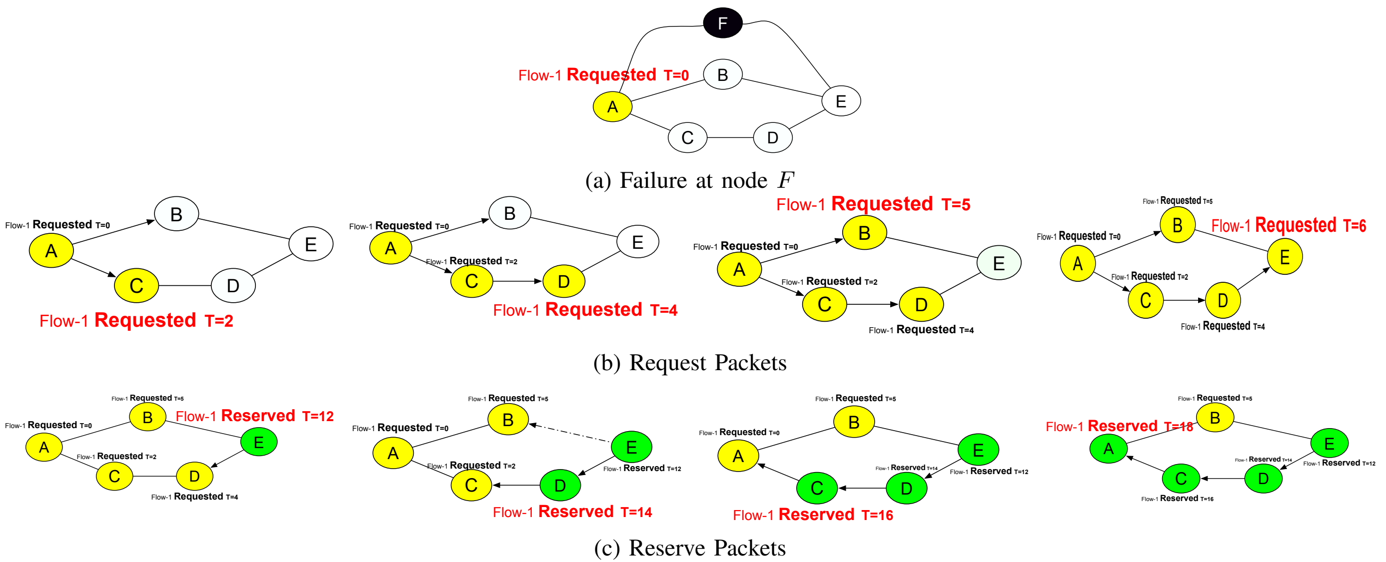
We use different types of routing packets in different phases of the routing algorithm. These types are $explore$, $request$, $cancel$, and $reserve$ routing packets. The functionality of $explore$ packets is to collect the information of node candidacy when they are injected into the network. The candidate information is stored in the data section of the routing packets. To prevent infinite forwarding loops, the Time-To-Live (TTL) field of $explore$ packets is initialized to the number of nodes in a given network. An $explore$ packet is silently absorbed at a node either when that node is the target node of the packet or its TTL decreases to 0.

The functionality of $request$ packets is to notify a node candidate that a real-time flow requests to utilize this node to transmit the flow. The functionality of $reserve$ packets is to reserve the resources on a node candidate for a real-time flow. Thereafter, this flow can utilize that node to transmit its messages with a deadline guarantee. When RPS processes a $request$ packet, it stores the information of the corresponding real-time flow into a local table and marks the table record as $requested$. After the $request$ packet has requested resources on all the node candidates on the path and arrives at the target node, a $reserve$ packet is sent back to change the record status from $requested$ to $reserved$. Once a $reserve$ packet is sent out, the system can subsequently utilize that path to transmit the corresponding real-time flow. The objective of $cancel$ packets is to notify a candidate node that the resources requested along this path have been canceled due to conflicts caused by routing packets for another real-time flow with higher priority.

Example: Fig. 3 shows this for flow-1: E-F-A. When node F fails, failure recovery to derive a new path between nodes E and A is triggered at node A (the destination of the flow). Node A, after marking the resources for flow-1 as $requested$ at time 0, sends $request$ routing packets. These $request$ packets subsequently mark the resources on nodes C, D, B, E as $requested$ for flow-1 at time 2, 4, 5, 6, respectively (shown in Fig. 3b). Then, since the routing packet from node D arrives earlier on node E relative to the routing packet from node B, node D, after reserving its resources for flow-1 at time 12, sends a $reserve$ routing packet back to node D, which traverses backwards and reserves the resources on nodes D, C, A for flow-1 at time 14, 16, 18, respectively (shown in Fig. 3c). A new real-time path E-D-C-A is established for flow-1.

The resources requested for flow-1 on node B must eventually be released so that they can be used to resolve requests of other real-time flows. In order to do this, when a node processes a $request$ packet, the node sets up a timer in its local table for the requested resources. We use notation T_2 to represent this timer. When T_2 for flow-1 expires at node B, the resources requested for flow-1 are released (i.e., the request record is removed from the local table of the node). This avoids the overhead (and bandwidth) of $cancel$ packets.

Multiple Failures: Gaps exist between the time when resources on a node are requested and when they are eventually reserved or canceled since these routing packets are processed in a distributed fashion (e.g., the resources on node C are requested at time 2 while they are reserved at time 16 in Fig. 3). Resource conflicts could happen during these time gaps. Notice that our approach is agnostic of the cause of conflicts, i.e., failure of a node common to two paths vs. failure

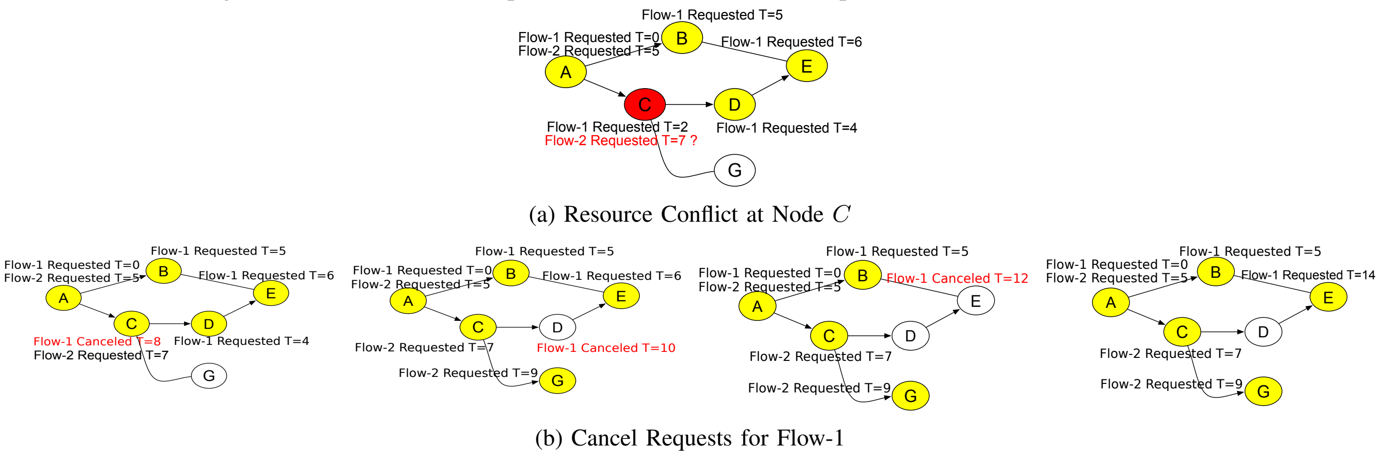


(a) Failure at node F

(b) Request Packets

(c) Reserve Packets

Fig. 3: Flow-1 A-F-E disrupted due to failure of F, new path established via A-C-D-E.



(a) Resource Conflict at Node C

(b) Cancel Requests for Flow-1

Fig. 4: Higher prio request for flow-2 from A to G causes conflict at C with lower prio request for flow-1 after F failed (as before); to resolve, cancel flow-1 request A-C-D-E before its path is reserved; flow-2 acquires A-C-G before flow-1 gets A-B-E.

of two different nodes on two paths are handled in the same manner. Fig. 4a depicts one such case for a conflict between flow-1 (see above) and flow-2, where the latter is requesting a path from A to G. A $request$ packet to find a new path for flow-2 arrives at node C at time 7. Node C must process this $request$ packet according to the priority of both flows. Had flow-1 had a higher priority, the $request$ packet for flow-2 would have been discarded at node C since the requirement for flow-1 should be met first. But this is not the case. Instead, flow-2 has a higher priority, so the resources requested by flow-1 are canceled since they have not been reserved yet (no green path established yet). Thus, $cancel$ routing packets are injected into the network by node C to cancel the requests for flow-1. Fig. 4b depicts the cancellation of the requested resources for flow-1 on nodes C, D, and E at time 8, 10, 12, respectively. Subsequently, node E sends a $reserve$ packet for the path A-B-E for flow-1 when it receives the $request$ packet from node B. Node G sends a $reserve$ packet for the path A-C-G for flow-2.

In Fig. 3c, the $reserve$ packet for flow-1 is processed by node E at time 12 and by node A at time 18. The requested resources for flow-1 must not be canceled during this period. Otherwise, the $reserve$ process cannot succeed. We adopt a timer T_1 to guarantee this. When a node processes a $request$ packet, the node sets up timer T_1 in its local table for the requested resources. When T_1 expires at a node, the resources requested for that flow on the node become exclusive. No other $request$ packets even with a higher priority can cancel exclusive resources. To guarantee that the $reserve$ process can always succeed, the target node will not send back a $reserve$ packet corresponding to a $request$ packet until T_1 has expired. To avoid exclusive resources being released due to the expiration of timer T_2, the time difference, $T_2 - T_1$, must exceed the aggregate processing time for the $reserve$ packet on every node on the path. The resources in either exclusive or reserved state can be utilized to transmit the corresponding real-time flow.

Fig. 5 summaries the resource state transitions during the

failure recovery process. Timer T_3 is used to change from the reserved state back to the available state when the node has not utilized these resources to transmit any message from the corresponding flow once the timer expires due to network failures. T_3 is reset to a value according to the period of the flow (e.g., twice the flow period in our implementation) whenever the node processes a message from that flow.

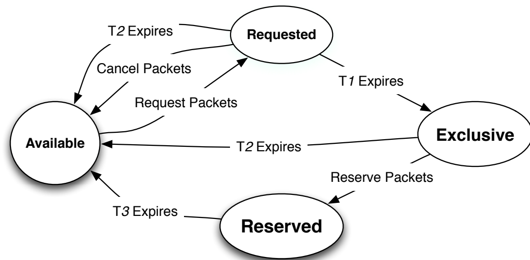


Fig. 5: Resource State Transitions

D. Failure Recovery Policy

With the failure recovery mechanism presented in Section II-C, we adopt a one-way failure recovery policy for Class-B flows and an alternate-path recovery policy for Class-A flows. The objective of these policies is to provide a fast recovery for Class-B flows, which are awaiting a new path to retransmit their messages, while reducing the total amount of the resources that are marked as $exclusive$ at the same time in the network, since the exclusive state prevents the resources to be reserved by another flow with even higher priority.

The one-way recovery for a flow f in Class-B consists of the following steps:

(1) When the destination node of f does not receive a message for f at the expected arrival time, it injects $request$ routing packets on the network. These packets carry the ID of the missing message and the flow, which is used as the unique identifier for this instance of failure recovery.

(2) When an intermediate node processes a $request$ packet and flow f passes the acceptance test on the node, the node associates the incoming port of the packet with the requested resources and stores this information in the local resource table. When the requested resources are associated with just one incoming port, the $request$ packet is forwarded to all output ports. Otherwise, no $request$ packet is forwarded (to avoid duplicates).

(3) When a $request$ packet arrives at the source node of flow f at time t, the source node creates a corresponding $reserve$ packet and marks it pending with an expected response time $t + T_1$. If a $cancel$ packet for this path arrives at the source node before $t + T_1$, the $reserve$ packet is ignored when it is processed. Otherwise, $reserve$ is sent to the destination node via a path constructed from the incoming ports associated with the resources. When multiple incoming ports are associated with the resources on any node, the port first associated is chosen to forward the $reserve$ packet to the next node.

(4) The source node can immediately utilize the path to transmit messages once a $reserve$ packet is sent. Since real-time messages and the $reserve$ packet are processed by different servers in the scheduler, a node can utilize the resources in exclusive state to transmit real-time messages. These resources are eventually reserved when the $reserve$ packet is processed. Thus, the transmission of the $reserve$ packet concludes this failure recovery instance. No further routing packets for the same instance are required to be processed.

When network conflicts occur on any node during this process, that node must inject $cancel$ packets to remove the corresponding resource requests. When an intermediate node processes a $cancel$ packet, it removes the incoming port of the packet. If no more incoming ports are associated with the requested resources, the node removes the request information from the local table and forwards the $cancel$ packet to all output ports. Otherwise, the request information is kept in the local table and no $cancel$ packets are forwarded.

Class-A flows utilize a two-phase recovery protocol. Let a Class-A flow have an initial set of n alternate paths with a requirement to maintain at least k alternative paths, where $k < n$. Since Class-A messages are multicasted along the set of alternative paths, the destination node can count the number of redundant messages received. We distinguish these two cases:

(1) If a Class-A flow degrades to p paths, where $k \leq p < n$, then the same one-way failure recovery protocol is initiated by the destination node, yet with a $lower$ priority than Class-B routing requests for path discovery. After all, with p the flow still has k or more paths and is not disrupted.

(2) If only p' paths with $p' < k$ remain or if a destination node D of such a flow does not receive a message at the expected time, then the recovery protocol is re-issued with a $higher$ priority than Class-B routing requests. Should lower priority requests be encountered on routing paths during this discovery, then these requests are upgraded to the $higher$ priority and $cancel$ requests are issued for lower requests that have not been confirmed yet by a reservation.

This policy integrates both classes of flows under a common recovery protocol, yet allows prioritization of recovery requests decoupled from the priorities of regular message delivery. And while we distinguish only Class-A/B flows here, a hierarchy of classes with a matrix of specific priorities per class and per redundancy level $p \in \{0..n\}$ could be supported.

E. Timing Analysis

Considering the limitations of the network resource capacity and the unpredictability of network failures, our mechanism cannot provide a guaranteed service that always derives new forwarding paths for all real-time flows affected by network failures. However, the choice of parameters α, β, T_1, and T_2 can be selected to ensure that a desired number of message flows can recover their paths. A larger value of α allows more routing packets to reside on a node at any time. β is the CPU quota replenishment rate for the routing packet server. $\frac{T_{rps}}{\beta}$ is the time for the server to accumulate sufficient quota to process one routing packet. T_1 determines the delay for a $reserve$ packet that is sent back to select the new path. T_2 determines

the delay before the exclusive resources become available for other flows. Thus, it determines the least inter-arrival time of network failures that our system allows.

When a network failure occurs, all flows that were transmitted via the failed nodes or links need to recover their paths. Our system provides a guaranteed upper bound for the recovery time of a flow in Class-B when network resources suffice and provides a best-effort recovery service for a flow in Class-A. For a flow $f(i)$ in Class-B, its recovery time $RT_B(i)$ can be upper bounded by the sum of (1) the aggregate processing time on each node on the path that transmits the $request$ packet for the corresponding final $reserve$ packet, (2) the aggregate propagation delay on the links of that path, and (3) delay T_1 before sending the final $reserve$ packet. Eq. 1 expresses this upper bound.

$$RT_B(i) \leq TD_B(i) + T_1, where$$
$$TD_B(i) = \sum_{0 \leq j < n(i)} (x(i,j) * \frac{T_{rps}}{\beta} + e) + L(i). \quad (1)$$

$TD_B(i)$ denotes the end-to-end transmission delay for the $request$ packet. $n(i)$ represents the number of nodes in the path of the $request$ packet. $x(i,j)$ indicates the number of routing packets for higher priority flows processed by node j in the path before processing $f(i)$'s $request$ packet. Thus, $x(i,j) * \frac{T_{rps}}{\beta}$ is the time for the node to replenish the process quota for $f(i)$'s $request$ packet. e is the execution time of the task in Algorithm 1 (including the routing packet processing time T_{rps} for the $request$ packet). $L(i)$ is the aggregate propagation delay on that path.

The following algorithm determines $x(i,j)$ in Eq. 1 offline. Given a network failure, the affected flows can be sorted increasingly by their relative deadlines, $f(0), f(1), ..., f(s-1)$, where s is the number of affected flows. If two flows have the same relative deadline, the flow with a smaller id exists first in the sorted sequence. Thus, i flows have shorter deadlines than flow $f(i)$. Since the packet scheduler processes routing packets in an EDF order, the routing packets of flow $f(i)$ need to wait for the routing packets of the previous i flows to be processed first on the same node. Given a flow with a higher priority $f(k)$, where $0 \leq k < i$, the jth node in our system can receive up to $k+1$ $request$ packets and $k+1$ $cancel$ packets on any one of its incoming links that the destination node of flow $f(k)$ can reach during the failure recovery. Let $p(j,k)$ be the number of those incoming links. Thus, the total number of routing packets of all flows with higher priorities than flow $f(i)$ is upper bounded by $\sum_{0 \leq k < i}(2 * (k+1) * p(j,k))$. In addition, since the packet scheduler is non-preemptive, the routing packet of flow $f(i)$ can potentially wait for the process of one routing packet of any flow with a lower priority. Eq. 2 summaries the upper bound for $x(i,j)$.

$$x(i,j) \leq 1 + 2 * \sum_{0 \leq k < i} (k+1) * p(j,k). \quad (2)$$

Furthermore, we adopt the following algorithm to determine the upper bound for the number of nodes, the corresponding propagation delay of the path, and the number of incoming links on every node for every affected flow (i.e., to determine $n(i)$ and $L(i)$ in Eq. 1 and $p(j,k)$ in Eq. 2).

(1) Remove the network components (links or nodes) assumed to have failed from the network topology. Sort the affected flows increasingly by their relative deadlines into flows $f(0), f(1), ..., f(s)$. Perform steps (2)-(5) for every flow $f(i)$ in the sorted order.

(2) Perform a depth-first traversal of the network graph starting from the destination of flow $f(i)$. For every node this traversal is visiting, we conduct an acceptance test for flow $f(i)$. If flow $f(i)$ fails the acceptance test on a node or that node is the source node of flow $f(i)$, the traversal does not search deeper. A node is marked as a candidate for flow $f(i)$ if $f(i)$ passes the acceptance test on that node. The resources that flow $f(i)$ requires on candidate nodes are excluded in the acceptance test for subsequent flows.

(3) The depth-first traversal simulates the forwarding process for routing packets of flow $f(i)$. Thus, the number of incoming links for flow $f(i)$ on every node can be determined after the traversal.

(4) Find the path from the destination to the source node of flow $f(i)$ of the longest recovery time $RT_B(i)$. The path must contain only candidate nodes for flow $f(i)$. A modification of Dijkstra's shortest path algorithm can achieve this.

(5) $n(i)$ and $L(i)$ are calculated as the number of nodes and the propagation delay on the path found in step (4).

The upper bound for the recovery delay in Eq. 1 can only be guaranteed if no routing packets for flow $f(i)$ are dropped and no resource requested by a flow with a lower priority becomes exclusive. Thus, the buffer reserved for routing packets on every node must be greater or equal to the aggregate size of all routing packets for $f(i)$ and all other flows with higher priorities. The transmission delay for the $request$ packet must not exceed T_1 as expressed by Eq. 3.

$$x(i,j) * size \leq \alpha * B[j] \wedge TD_B(i) \leq T_1. \quad (3)$$

Eq. 3 can check if the upper bound for the recovery delay for a flow in Class-B is guaranteed for a given α, β, and T_1. We next evaluate this upper bound experimentally.

F. Discussion

We first discuss under what conditions a high priority flow is guaranteed to derive a new path if sufficient resources exist for this flow but other lower priority flows are simultaneously trying to find new paths. Let us assume that all flows have disjoint priority levels. We can then distinguish two cases:

(1) If a resource conflict beyond network capacity exists at a node during path recovery for two or more flows, then the high priority flow is guaranteed to recover while the lower priority one(s) beyond capacity will not find an alternate path. This is implied by the upper bound for recovery delay of a flow established in Sec. II-E. As stated, if the buffer reserved for routing packets on every node is greater or equal to the aggregate size of all routing packets for $f(i)$ and all other flows with higher priorities, then Eq. 3 can be used to check if the upper bound of Eq. 1 can be satisfied for the selected (T1, alpha, beta) values. Within this upper bound, a new path will be found by the high priority flow if one exists.

(2) If no resource conflict exists for some (low priority) flow, then this flow may recover a path before any other (high

priority) flow. This situation can only happen if the low priority flow finds a path p' and *reserves* it prior to the path discovery for the high priority flow reaching any nodes of p'. To prevent such recovery races, T_1, the delay on the source node, can be set to exceed the propagation delay of *reserve* requests along *any* possible recovery path for higher priority flows, e.g., 50ms for a 5ms propagation delay per hop and a maximum of 10 hops.

(3) We next argue that the state diagram in Fig. 5 prevents deadlock. If a node fails, any node along this path in *reserved* state will eventually give up its resources and become *available*, namely when timer T_3 expires (since the node could not establish a route to transmit a given message associated with this path and timer). Also, any nodes along such a path in *exclusive* or *requested* state will eventually give up its resources and become *available* when T_2 expires. Furthermore, any node in *requested* state will give up resources if its flow does not use it within a delay of T_1 from entering this state. Hence, any inactivity due to failures results in the release of resources. We further argue that a livelock cannot occur under the conditions described in (1) and (2).

III. Evaluation

A. Evaluation Platform

We have implemented a virtual switch based on the structure of commercial physical switches [1]. This virtual switch implements a store-and-forward model with packet queues at output ports. A packet arriving at any input port is first put into shared memory on the switch. Then, the routing processor processes the packet to determine the forwarding rule and forwards the packet to the output queue of the corresponding output port. Fig. 6 depicts the structure of such a virtual switch. The input queue (marked gray) serves as a means to simulate propagation delays on network links. A packet is moved from the input queue to shared memory only after the simulated delay of that packet has passed.

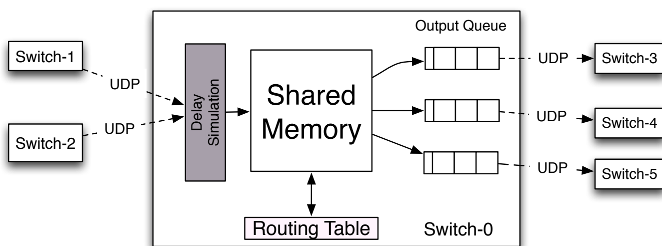


Fig. 6: Virtual Switch Structure

Each virtual switch is executed in a user-level process on Linux and utilizes two *Posix threads* to simulate the behavior of input/output ports and the routing processor, respectively. Virtual switches use *UDP* transmissions to simulate network links. The first thread, the so-called "port thread", is responsible for message transmissions. It uses the $recvfrom$ system call to receive messages and puts them into the input queue. It then moves packets into shared memory after their simulated delay has passed. The port thread also uses the $sendto$ system call to forward messages in the output queues to the next hop. The second thread, the "routing thread", is responsible for message processing. It determines the routing policy via the local routing table and moves real-time and background messages into the corresponding output queues. In addition, the routing thread responds to routing packets according to the failure recovery policy presented in Section II-D. As a result, the routing thread modifies the local routing table to support dynamic failure recovery.

We evaluate our protocol on this virtual platform on a cluster, where each node features 2 AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node) with 32GB DRAM and Gigabit Ethernet. Reported results are the average over 10 runs with a negligible standard deviation in our virtual switch environment.

B. Setup-1

Configuration: We isolate one switch node to simulate a network failure and measure the end-to-end response time of real-time flows before and during this failure. In addition, we measure the recovery time for flows whose forwarding paths are disrupted by the failure. Since our approach is agnostic of the cause of conflicts, failures of flows crossing a common failed node are handled the same as failures of two different nodes, each affecting disjoint flows, i.e., the following scenarios cover multiple simultaneous failures.

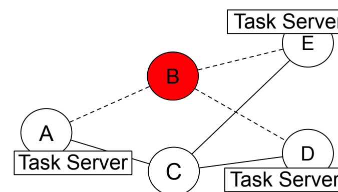


Fig. 7: Failure Recovery Setup (1)

In this experiment, switches A - E are interconnected as shown in Fig. 7. One task server is attached to each of the switches A, D, and E to generate transmission workloads. Table I presents the temporal properties and forwarding paths of these transmissions before the network failure occurs. All flows are Class-B real-time flows and have the same phase. Switch B fails $10s$ after the system starts. Then, the failure recovery procedure for flows 1 - 3 is triggered at the task server on their destinations. We add a $5ms$ delay on the UDP transmissions to simulate the propagation delays on physical links in wide-area networks. We provide a sufficient buffer size for routing packets (i.e., α is sufficient) in this experiment. Then, the value of β is changed and we measure its impact on the transmission times of packets for real-time flows 4 - 8 and the recovery time for flows 1 - 3.

Results: The experimental results show that the end-to-end response time for real-time flows before network failures are in the range $[10.1ms, 10.6ms]$. During failure recovery for flows 1 - 3, the worst case end-to-end response time for other flows increases to $10.9ms$. The non-preemptive procedure in lines 26 - 30 in Algorithm 1 (i.e., T_{rps}), which processes one routing packet at a time, contributes to the increase of the response

TABLE I: Workload (1)

Flow	Period (ms)	Relative Deadline (ms)	Forwarding Path
1	55	80	A→B→D
2	55	90	A→B→D
3	55	100	A→B→E
4	40	100	D→C→A
5	50	100	D→C→A
6	40	100	A→C→D
7	50	100	A→C→D
8	60	100	A→C→E

time for real-time packets. The measured T_{rps} is $0.4ms$ in our system. This overhead is constant, i.e., it is unaffected by changes in β.

Observation 1: Our failure recovery protocol can delay the transmission of a real-time packet by up to time T_{rps} on each switch on the forwarding path, where T_{rps} is the worst-case processing time for one routing packet.

Fig. 8 depicts the worst-case failure recovery time for flow 1 - 3 when β is changed. For example, when β is 0.10 (i.e., up to 10% CPU time can be used to process routing packets on each switch), the worst-case recovery time is $10.2ms$ for flow 1, $27.9ms$ for flow 2, and $28.6ms$ for flow 3. Since the accumulated quota rpq at the time of a failure is sufficient to process one routing packet, the $request$ packet for flow 1 can be processed immediately when it arrives at switches D, C, and A, respectively. Thus, the recovery time for flow 1 is just the one-way transmission time equivalent to a regular real-time packet. This recovery time remains constant even when β is changed (depicted by the green line in Fig. 8). However, the routing packets for the highest priority flow could experience a delay of up to $\frac{T_{rps}}{\beta}$ on each switch on the path if the processor is executing the non-preemptive procedure (lines 26 - 30 of Algorithm 1) for another flow with lower priority.

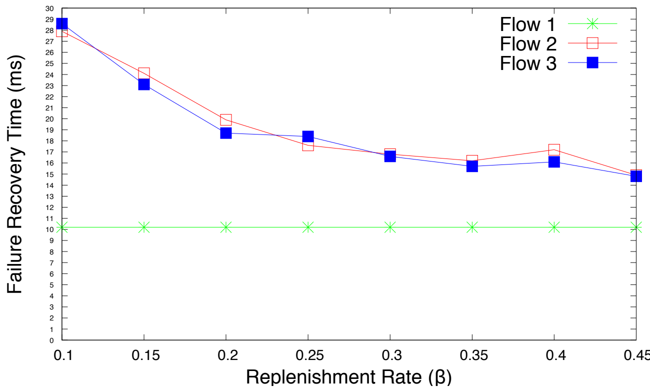


Fig. 8: Failure Recovery Time Results

The $request$ packets of flow 2 share the same forwarding path with the $request$ packets of flow 1. Thus, the processing quota rpq has to be replenished before any $request$ packet for flow 2 can be processed. This contributes a delay of up to $4ms$ (i.e., $\frac{T_{rps}}{\beta}$) on each switch when β is 0.10. The measured value for e in Eq. 1 is $2.0ms$ in our system. In addition, no $cancel$ packets are generated due to resource conflicts in this setup. $x(i,j)$ can be upper bounded by $1+\sum_{0\leq k<i} p(j,k)$ in Eq. 2, which is 2 for flow 2 on each switch. Thus, $TD_B(i)$ in Eq. 1 derives a safe upper bound for the recovery time for flow 2. This even holds when β is changed (see Fig. 8).

Flow 3 has a similar (even shorter) recovery time compared to flow 2, even though its routing packets have a lower priority. This is due to flow 3 using a different forwarding path for its $request$ packet, which is E→C→A. When its $request$ packet arrives at switch A, the quota rpq has been partially replenished on A. Thus, the quota does not have to wait for an entire $\frac{T_{rps}}{\beta}$ round to become sufficient to process the $request$ packet for flow 3. However, the recovery time for flow 3 shows the same decreasing trend when β is increased.

Observation 2: The recovery time for any flow decreases when β increases, since this increases the replenishment rate for the processing quota of routing packets.

However, $1-\beta$ is the maximal system utilization for real-time packet transmissions. Thus, given a specific task set, one can determine the system utilization when the static routing algorithm is performed and determine the maximal value for β, which provides the largest replenishment rate for the processing quota of routing packets.

C. Setup-2

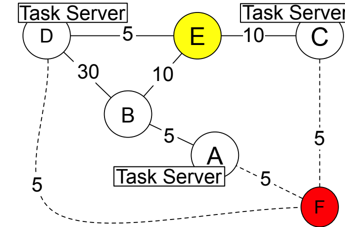


Fig. 9: Failure Recovery Setup (2)

Configuration: We create a complex network topology and analyze the routing process for failure recovery. Switches A - F are connected as shown in Fig. 9. Table II presents the temporal properties and forwarding paths of these transmissions before and after the network failure occurs at time t=0, which disconnects switch F. Numbers on the edges in Fig. 9 indicate the propagation delays of the corresponding links (in ms). We limit the buffer size on switch E so that only one flow can utilize it to forward its packets. This subsequently demonstrates the functionality of $cancel$ packets. Here, $\beta = 100\%$ so that the routing packets can be processed once they arrive at a switch in this experiment.

TABLE II: Workload (2)

Flow	Period	Relative Deadline	Forwarding Path	Alternate Path
1	50	100	A→F→D	A→B→D
2	50	80	A→F→C	A→B→E→C

Analysis: The task servers start the failure recovery procedure for both flows at the destination node (i.e., node D for flow 1, node C for flow 2) since we set the delay to trigger failure recovery to T_2 = 200ms in this experiment. Fig. 10 depicts the routing process. Blue, red, or yellow circles indicate that the resources on the corresponding node are requested for flows 1, 2, or both flows, respectively. Solid

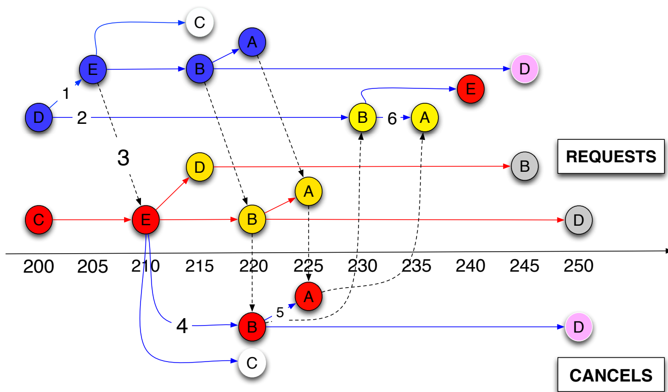


Fig. 10: Failure Recovery Timeline

blue and red edges indicates a routing packet transmission for flows 1 and 2, respectively. For example, the edges labeled 1 and 2 represent the events that node D transmits one $request$ packet to each of the nodes E and B at time 200. A dotted edge denotes a change of resource request state on a node. For example, edge 3 indicates that the resources requested by flow 1 on node E at time 205 are seized by flow 2 at time 210 due to the resource conflicts on E and since flow 2 has a shorter relative deadline (i.e., higher priority). As a result, $cancel$ packets are sent to cancel the requests for flow 1 on nodes B and A (edges 4 and 5). However, the resources on nodes B and A are requested for flow 1 again at time 230 and 235 due to the processing of the routing packet on an alternate path $D \rightarrow B \rightarrow A$ (edges 2 and 6).

In addition, Fig. 10 demonstrates the cases when a routing packet is not forwarded as follows. (1) White circles: the corresponding nodes have no outgoing ports for the arriving routing packets. (2) Pink circles: the corresponding nodes are the destinations of the flows. (3) Grey circles: the flows have multiple incoming ports at the corresponding nodes.

The task server on A ultimately sends $reserve$ packets to reserve the forwarding path $A \rightarrow B \rightarrow D$ for flow 1 and $A \rightarrow B \rightarrow E \rightarrow C$ for flow 2.

Notice that scenarios with Class-A flows can be reduced to two cases depending on how many alternative paths p remain. If sufficient paths $k \leq p$ remain, Class-B recovery is only impacted minimally by T_{rps} per switch (see Observation 1). If $p < k$ paths remain, Class-A recovery requests impose a blocking term on Class-B recovery. In general, this blocking term b is bounded by

$$b = \sum_{i \in hp} RT_B(i) + TD_B(i), \qquad (4)$$

where hp denotes the set of higher priority flows, each of which requires a recovery cost of the end-to-end transmission delay RT_B for requests, issuing the initial reserve packet delay $T1$ (included in RT_B), and TD_B for its propagation back, see Equation (1). If a priority matrix is used, the upper bound is conservatively given for $k = 0$ (first column), or bounds can be computed separately for each k-level in analogy to mixed-criticality systems.

Observation 3: Recovery overhead can be upper bounded analytically by the aggregate of a round-trip end-to-end transmission delay over all higher priority flows.

In the example of Setup-2, flow 1 issues a request at time 200 simultaneous to flow 2. By considering only the propagation delay $L(i)$ in Equation (1) and assuming sufficient server capacity, the longest request for recovery path of flow-2 is $C - E - D - B - A$ with length $10 + 5 + 30 + 5 = 50$, or a round-trip total of $50 + 50 = 100$. This is an upper bound for the actual delay of $C - E - B - A$ and back for a total of $25 + 25 = 50$. Notice that the longer request via D is silently absorbed at node B by our protocol.

In Fig. 10, only request but not reserve packets are depicted. Flow 2 discovers its alternate path at time 225 (and reserves it at 250, now shown). Flow 1 receives cancellations until 225 on its attempt to reserve a path via E, but the simultaneous request to B is granted at time 230 and makes it to the source node A at 235. Hence, the actual blocking term for flow 1 is zero in this case. Hence, our bound holds, even though it appear conservative in this case. Optimizations are possible by considering overlapping paths (see silent absorption above) but would be hard to handle for pairwise path dependencies in the general case.

IV. Related Work

Our failure recovery mechanism establishes new real-time forwarding paths dynamically when network failures have disrupted the old paths. Past work has proposed different mechanisms to establish communication channels to support the real-time requirements of data transmissions. Our work differs as follows. First, our mechanism considers potential resource conflicts when multiple tasks intend to recover their paths at the same time. Past work does not consider the impact of these conflicts [3], [5], [2], [6] or only provides network resources to the requests on a first-come-first-served basis [14]. Instead, our mechanism allocates resources to the transmission with highest priority first, which resolves the problem discussed in Section I. Our work is more suitable when failures disrupt multiple real-time channels.

Second, past work utilizes centralized or hierarchical schedulers to govern the admission of real-time flows that need to reserve resources for their transmissions [11], [7]. Our mechanism is more robust as it is fully distributed. While the failure recovery procedures are triggered by the destination nodes of the corresponding real-time transmissions, the network devices progress autonomously to establish new paths. To achieve this, we store resource states in a local table on each network device, which is similar to D^2TCP [5].

Third, our mechanism provides deadline guarantees to real-time transmissions at the same time when the failure recovery is in progress. This differs from past work that utilizes either rate-controlled service disciplines [4], [16] or EDF scheduling [5], [17] without dynamic rerouting reservations. We combine the benefits of an EDF-based scheduler for real-time packet transmissions [9] with a routing packet server on the basis of the Total Bandwidth Server [12] to process routing requests. The goal is to prevent deadline misses for real-time transmissions caused by the interference of failure recovery.

Our work derives disjoint forwarding paths for real-time flows with short relative deadlines (i.e., in Class-A).

We believe that disjoint path algorithms based on network graphs [13], [15] can be extended to establish new paths required for real-time constraints.

V. Conclusion

We have presented a failure recovery mechanism to dynamically establish new forwarding paths for real-time transmissions in a distributed computing environment. Our mechanism considers the priorities of the corresponding real-time tasks when resource conflicts occur during the failure recovery process, which allocates resources to the transmission with higher priority first. In addition, we bound the computation capability and size of the device buffer of the server dedicated to processing dynamic routing requests to prevent interference of real-time transmissions. Thus, deadlines can be guaranteed to real-time transmissions when failure recovery is in progress. We have implemented a virtual platform and the failure recovery mechanism to evaluate our approach. The virtual platform utilizes Linux user-level processes and threads to simulate the behavior of network devices. It utilizes UDP to simulate data transmissions over network links. We have evaluated this system on a local cluster to assess the effectiveness of our mechanism.

References

[1] Catalyst switch architecture and operation. http://www.cisco.com/networkers/nw03/presos/docs/RST-2011.pdf.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] Tao Qian, Frank Mueller, and Yufeng Xin. A linux real-time packet scheduler for reliable static sdn routing. In *Real-Time Systems (ECRTS), 2017 29th Euromicro Conference on*, June 2017.

[10] Parameswaran Ramanathan and Kang G Shin. Delivery of time-critical messages using a multiple copy approach. *ACM Transactions on Computer Systems (TOCS)*, 10(2):144–166, 1992.

[11] 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.

[12] Marco Spuri and Giorgio C Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *RTSS*, pages 2–11, 1994.

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

[14] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[15] 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.

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

[17] 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.

Tao Qian is a computer systems engineer. He received his Ph.D. from North Carolina State University in 2017. He has published papers in the areas of distributed systems, embedded and real-time systems.

Frank Mueller (M94-SM06-F16) is a Professor in Computer Science and a member of multiple research centers at North Carolina State University. Previously, he held positions at Lawrence Livermore National Laboratory and Humboldt University Berlin, Germany. He received his Ph.D. from Florida State University in 1994. He has published papers in the areas of parallel and distributed systems, embedded and real-time systems and compilers. He is a member of ACM SIGPLAN, ACM SIGBED and a senior member of the ACM and an IEEE Fellow. He is a recipient of an NSF Career Award, an IBM Faculty Award, a Google Research Award and two Fellowships from the Humboldt Foundation.