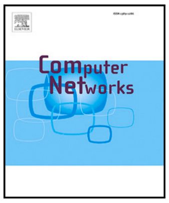
Computer Networks 235 (2023) 109983

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Reinforcement learning based routing for time-aware shaper scheduling in time-sensitive networks[☆]

Junhong Min [a], Yongjun Kim [a], Moonbeom Kim [a], Jeongyeup Paek [a,*], Ramesh Govindan [b]

[a] Department of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea
[b] Department of Computer Science, University of Southern California Los Angeles, CA, USA

ARTICLE INFO

Keywords:
Time-Aware Shaper (TAS)
Time-Sensitive Network (TSN)
Reinforcement learning
Routing
Scheduling
Network simulation
Network performance evaluation

ABSTRACT

To guarantee real-time performance and quality-of-service (QoS) of time-critical industrial systems, time-aware shaper (TAS) in time-sensitive networking (TSN) controls frame transmission times in a bridged network using a scheduled gate control mechanism. However, most TAS scheduling methods generate schedules based on pre-configured routes without exploring alternatives for better schedulability, and methods that jointly consider routing and scheduling require enormous runtime and computing resources. To address this problem, we propose a *TSN Scheduler with Reinforcement Learning-based Routing (TSLR)* that identifies improved load balanced routes for higher schedulability with acceptable complexity using distributional reinforcement learning. We evaluate *TSLR* through TSN simulations and compare it against state-of-the-art algorithms to demonstrate that *TSLR* effectively improves TAS schedulability and link utilization in TSN with lower complexity. Specifically, *TSLR* shows a more than 66% increase in schedulability compared to the other algorithms, and *TSLR*'s scheduling time is reduced by more than 1 h. It also shows flows' transmission latency is less than 25% of their latency deadline requirement and reduces maximum link utilization by approximately 50%.

1. Introduction

Time-Sensitive Networking (TSN) [1,2] represents a general-purpose real-time Ethernet standard[1] that aims to solve not only the real-time requirements but also the compatibility issues of many proprietary Ethernet extensions such as EtherCAT, PROFINET, and SERCOS III. Its goal is to provide a standards-based deterministic ultra-low latency, ultra-low jitter, and zero-congestion loss data communication in an integrated network that supports both time-sensitive and best-effort traffic simultaneously. TSN yields a next-generation local area network (LAN) technology for the coexistence of information technology (IT) and operation technology (OT), especially for industrial automation, in-vehicle, and avionic networks.

TSN consists of several standards to ensure the stringent timing requirements of real-time systems. Particularly, the IEEE 802.1Qbv [3] serves as one of the TSN's core standard amendments defining the *time-aware shaping (TAS)* mechanism, which aims to schedule precise frame transmission timing in bridged networks. TAS guarantees deterministic latency for time-sensitive traffic flows by controlling the transmission gates of egress queues within switches according to computed schedules (Fig. 1). However, calculating correct and coordinated TAS schedules for a network poses a challenging problem and requires substantial computation because it must account for various factors such as traffic configuration, flow requirements, paths, link capacity, and utilization [4,5]. Optimal TAS scheduling is known to be an NP-complete problem [6,7].

Problem. Prior works have proposed constraints and methods for TAS scheduling (related works in Section 2). Many of these works focus on scheduling based on the assumption that the routes of the input flow are known in advance [8–14]; they do not explore the dependency between routes and schedules. Nevertheless, if a flow passes through a path, that affects the schedule of other flows on that path. A set of

☆ This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A2C1008840), also by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP2023-RS-2022-00156353) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and also by the Chung-Ang University Graduate Research Scholarship in 2021.

* Corresponding author.

E-mail addresses: dmc93@cau.ac.kr (J. Min), healu1423@cau.ac.kr (Y. Kim), mbkim@cau.ac.kr (M. Kim), jpaek@cau.ac.kr (J. Paek), ramesh@usc.edu (R. Govindan).

[1] To be precise, TSN does consider (and partially support) other layer-2 technologies [28], but the main focus is still on Ethernet.

https://doi.org/10.1016/j.comnet.2023.109983
Received 30 March 2023; Received in revised form 23 July 2023; Accepted 12 August 2023
Available online 18 August 2023

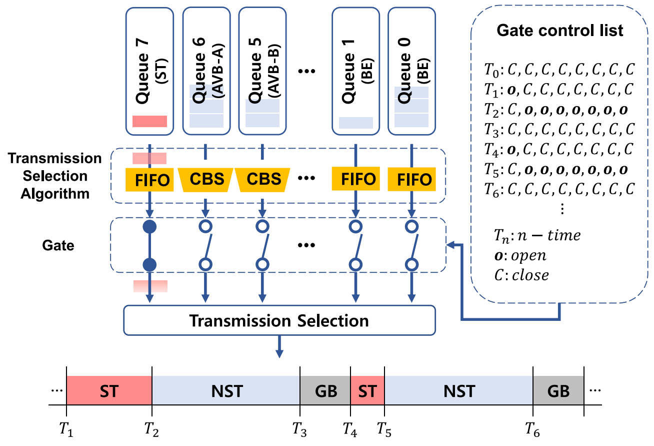

Queue 7 (ST)
Queue 6 (AVB-A)
Queue 5 (AVB-B)
Queue 1 (BE)
Queue 0 (BE)
Transmission Selection Algorithm
FIFO
CBS
CBS
FIFO
FIFO
Gate
Transmission Selection
Gate control list
T_0: C, C, C, C, C, C, C, C
T_1: o, C, C, C, C, C, C, C
T_2: C, o, o, o, o, o, o, o
T_3: C, C, C, C, C, C, C, C
T_4: o, C, C, C, C, C, C, C
T_5: C, o, o, o, o, o, o, o
T_6: C, C, C, C, C, C, C, C
T_n: n − time
o: open
C: close
ST
NST
GB
ST
NST
GB
T_1
T_2
T_3
T_4
T_5
T_6

Fig. 1. Illustration of IEEE 802.1Qbv time-aware shaper (TAS) and gate control list (GCL).

flows may not offer a valid schedule on a given topology according to the route configuration. For this reason, there may be an opportunity to improve schedulability if alternate routes can be explored, which are attempted in [15–20]. However, searching through all possible routes for each flow within a large network topology and then creating and comparing multiple schedules to identify the optimal presents intractable complexity and requires enormous runtime and computing resources resulting in low schedulability.

Motivation: Motivated by the complexity issues, we propose a novel *TSN scheduler with reinforcement learning-based routing (TSLR)*. For improved TAS schedulability, we tackle routing and scheduling as one problem; however, a brute-force *try-all* strategy cannot be scalable. As demonstrated in Section 5, the state-of-the-art routing and TAS scheduling algorithms in TSN literature fail to complete within a reasonable amount of time when the network and/or flow set size grows. To this end, we design *TSLR* based on the *Categorical Deep Q-Network (C-DQN)* [21] reinforcement learning algorithm. Our intuition is that the intelligent policies of reinforcement learning could help solve complex problems despite of partial explorations.

Approach: *TSLR* consists of two main components; *distributional reinforcement learning-based load-balanced routing* (*DRR*) and *path step scheduling* (*PSS*) algorithms. *PSS* creates TAS schedules efficiently on a per-link basis considering flow interval, deadline, link bandwidth, and path length. However, *PSS* may fail if the link utilization or flow deadline requirements cannot be satisfied. Furthermore, if the paths of flows are fixed, whether a valid set of schedules exist for all flows (i.e., schedulability) becomes deterministic. Thus, *DRR* dynamically explores alternate routes that can be scheduled for an identical set of flows. While doing so, *DRR* considers two factors; the first is the number of flows scheduled successfully while satisfying the deadline requirements, a *TSLR*'s primary objective. *TSLR* attempts to schedule as many flows as possible on the given topology by altering the routes. The second is load balancing. *TSLR* favors minimizing maximum link utilization to provide ample room for other flows (including non-time-critical best-effort traffic) and utilize network resources efficiently. To the best of our knowledge, there is no prior work that has used distributional reinforcement learning to generate load-balanced routing for improved TAS schedulability.

Contributions: This work's contributions are as follows;

- We propose *TSLR* that identifies improved load-balanced routes for higher schedulability by exploring alternatives.
- This is achieved with feasible complexity by applying reinforcement learning to routing and TAS scheduling for TSN.
- We evaluate *TSLR* through TSN simulations to exhibit improved schedulability and link utilization of TAS compared to several state-of-the-art algorithms in the literature.

The remainder of this paper is structured as follows. In Section 2, we first summarize related literature. Then, we introduce TAS and reinforcement learning, and justify the selection of C-DQN for our work in Section 3. We present the *TSLR* design in Section 4, and evaluate *TSLR* in Section 5. We summarize the work and conclude in Section 6.

2. Related work

Several prior studies have attempted to solve the problem of optimal or efficient TAS scheduling in TSN. Craciunas et al. [8] propose formal scheduling constraints for calculating valid GCL considering the influencing factors of real-time communication and computing schedules by applying satisfiability modulo theory (SMT). Dobrin et al. [9] propose a fault tolerance scheduling scheme that ensures the deadlines of time-sensitive traffic considering transmission failures and retransmissions. Jin et al. [10] employ SMT and optimization modulo theory (OMT) to schedule more real-time flows and reduce execution time. Ansah et al. [11] propose a schedulability analysis algorithm that verifies whether schedules of periodic TSN applications and bridges can be computed. Durr et al. [12] propose a heuristic scheduling algorithm based on the Tabu search and a compression algorithm to reduce bandwidth wastage by guard bands. Kai et al. [22] propose TSN Chained Flow Scheduling (TCFS) as an efficient scheduling mechanism in a multi-level topology. Their ILP-based approach is designed to resolve offline scheduling problems (typical TAS scheduling cases), and Tabu search based approach is designed to solve online scheduling problems. However, these works limit the search space by assuming fixed and given routes with simplified constraints without considering alternative routes for improved schedulability.

Some studies have considered routing jointly with scheduling in order to improve schedulability. Nayak et al. [23] propose a time-sensitive software-defined network (TSSDN) that exploits the logically centralized paradigm of SDN to provide ILP formulations for solving the combined problem of routing and scheduling time-triggered traffic. Subsequently, the authors improve performance by proposing an ILP-based incremental scheduling algorithm that dynamically adds schedules whenever new flows occur [15]. Similarly, two recent works [24,25] propose methods for dynamic (online) re-configuration of TAS scheduling and routing. However, our work aims to solve the offline scheduling and routing problem in TAS.

Schweissguth et al. [16] propose an ILP-based joint routing and scheduling method that formularizes network structure, routing, scheduling, and application requirements. Smirnov et al. [17] propose an approach that generates a valid route and schedule using a set of pseudo-boolean constraints for automated optimization of mixed-criticality networks with time-triggered traffic. Xu et al. [18] utilize SMT and OMT to solve the co-design constraint set of scheduling and routing in TSN. Alnajim et al. [19] propose a QoS-aware path selection and scheduling algorithm that calculates the route and schedule incrementally to minimize queueing delay and preserve QoS. Hellmanns et al. [20] propose an ILP-based routing and scheduling method, improving its performance through various optimization techniques. However, these works only handle simple constraints (simplification of requirements such as deadline, network load, topology, etc.), suffering scaling issues for more complex problems with a larger number of flows.

Most recently, there have been attempts to adopt reinforcement learning for TSN. Yang et al. [26] propose a Graph Convolutional Network-based routing and TAS scheduling scheme, and Yu et al. [27] propose a branching dueling Q-network-based scheme. However, they have not evaluated their proposals against ILP- or metaheuristic-based approaches, which are widely adopted in TSN. Furthermore, their

Table 1
Summary of related works.

Related works	Main difference from our work
[8–10,12,22]	Does not consider routing optimization.
[15,24,25]	Focus on online scheduling.
[16–20,23]	Does not use reinforcement learning.
[26,27]	Does not use distributional reinforcement learning.

evaluations are conducted with very simple network requirements. For example, all flows have the same and loose transmission interval [26], such as 5 ms, or only small network topologies under ten nodes are considered [27]. More importantly, none of these works have investigated applying distributional reinforcement learning to increase TAS schedulability.

Table 1 summarizes the list of most relevant prior works and their key differences from our work.

3. Background

We begin with an initial overview of TSN's *time-aware shaper* mechanism and *reinforcement learning*.

3.1. IEEE 802.1Qbv Time-Aware Shaping (TAS)

One of TSN's goals includes supporting a variety of traffic types in a converged network. These are generally classified into *scheduled time-critical (ST)*, semi time-sensitive *audio-video bridging (AVB)*, and *best-effort (BE)* traffic based on their requirements. An ST flow is a periodic flow that requires deterministic ultra-low latency (with hard deadlines) and low jitter without congestion loss. To guarantee these requirements, TAS in TSN isolates those flows' transmission times from other traffic types using a gating mechanism, which schedules the transmission gates of egress queues within each switch based on advanced knowledge of flow information.

Fig. 1 illustrates an example of TAS operation in a TSN switch. A switch can have up to eight queues in each egress port, and each queue corresponds to a *traffic class* determined from the *priority code point (PCP)* in the VLAN identifier according to the IEEE 802.1Q mapping [28]. Each queue may have an individual *transmission selection algorithm* (e.g., *credit-based shaper* (CBS) [29], *asynchronous traffic shaper* (ATS) [30], or a simple FIFO) that can throttle the transmissions based on some stream reservation criteria [31] (or lack thereof). Additionally, each queue presents a *transmission gate* that has two states, *open* or *closed*. Frames in a queue can be transmitted only through an open gate. When multiple gates are open simultaneously (e.g., at time T_1 in Fig. 1), the *transmission selection* part selects and transmits frames in descending order of *traffic class* among those open queues according to the strict-priority rule [14].

The gates are controlled by the *gate control list (GCL)*. To compute this GCL, TAS first allocates ST time windows that can accommodate the transmission times of time-critical flows that need to be transmitted at the given interval. Furthermore, to prevent delaying ST frames due to non-ST frames, TAS allocates a *guard band (GB)* that closes all gates before every ST window, as expressed at the bottom of Fig. 1. Finally, all remaining times except for the ST and GB windows are assigned as the *non-scheduled traffic (NST)* windows for transmission of all other traffic types such as AVB and BE.

3.2. Reinforcement Learning (RL)

RL aims to learn an agent's optimized behavior by taking actions to maximize reward in a specific environment.

Q-learning [32] represents an RL algorithm that uses *Markov Decision Process (MDP)* as a probability model (Fig. 2). In our work, input traffic flows become the agent in the MDP, and the current network state is

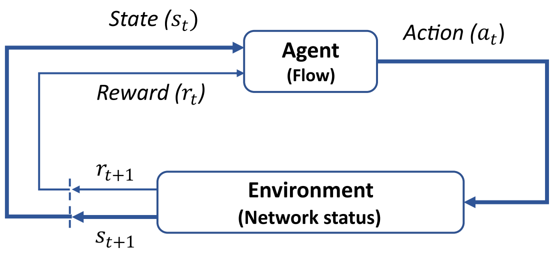


Fig. 2. Reinforcement learning Markov decision process.

the environment. In Q-learning, the agent learns the optimal policy by predicting an action's future reward value (Q-value) for a specific state in MDP. A Q-function that generates Q-value is expressed as:

$$Q_\pi(s_{t,}a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | s_t, a_t]. \tag{1}$$

When Q_π has a policy π, it returns the sum of expected rewards E, which can be obtained by action a_t in state s_t. R_{t+n} is the reward value that can be obtained in n time units, and γ is the discount rate that expresses how important the reward of the currently selected action is compared to a future reward. γ possesses a value between 0 and 1 and is designed such that the distant future reward yields a less effect than the present reward.

Initially, Q-function is initialized to an arbitrary value, and then learns using the following formula;

$$Q_{new}(s_t, a_t) = (1-\beta)Q(s_{t,}a_t) + \beta(r_t + \gamma max_a Q(s_{t+1}, a)) \tag{2}$$

β represents the learning rate which makes the change in Q faster as it approaches 1. $Q(s_{t,}a_t)$ is the current Q-value, r_t is the current action's reward value, and $\gamma max_a Q(s_{t+1}, a)$ is the maximum Q-value that can be obtained in the next state. Based on these, the new Q-value $Q_{new}(s_t, a_t)$ is updated. An iteration ends when the state reaches the terminal state (depending on the model's definition, it may not exist) or the process runs for a predefined number of iterations.

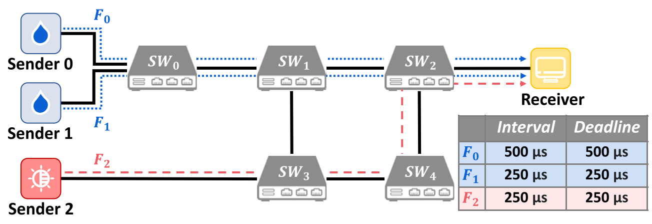
Deep Q-Network (DQN) [33]**:** Q-learning presented successful results in various domains but only for those with low-dimensional states. When actions and states are combined, many real-world problems become high-dimensional, and the massive data scale makes processing with Q-learning difficult. DQN solves the Q-learning data scale problem by learning to approximate Q-values through artificial neural networks. The DQN's model is expressed as $Q(s, a; \theta)$, where θ represents the weights to be trained in the neural network and the cost function is expressed as,

$$Loss = [Q(s, a; \theta) - (r(s, a) + \gamma max_a Q(s', a))]^2 \tag{3}$$

The squared difference between the predicted Q-value and the sum of present and future rewards equates to cost, and then learning proceeds such that the cost converges to zero.

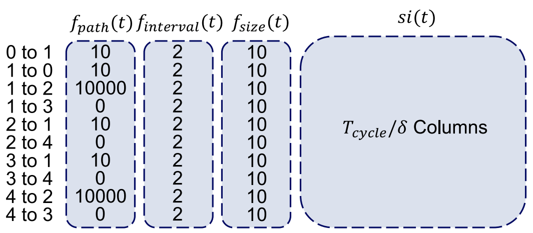
Categorical Deep Q-Network (C-DQN) [21] is the RL algorithm that we adopt for *TSLR*. Admittedly, there are many other RL algorithms from which we could choose. However, our intuition is that *C-DQN* is better suited for *TSLR* in solving the load-balanced routing and TAS scheduling problem of TSN due to the following reasons;

- C-DQN, while similar to a DQN, utilizes the Bellman equation to learn approximate value distributions.
  DQN's prediction as a single scalar value for a particular state of a complex environment with partial observations does not adequately reflect the variations in real systems.
  Therefore, expressing the reward as a distribution as in C-DQN could help to predict the future reward more accurately.



	Interval	Deadline
F_0	500 μs	500 μs
F_1	250 μs	250 μs
F_2	250 μs	250 μs

Fig. 3. Example for describing *TSLR*.



	$f_{path}(t)$	$f_{interval}(t)$	$f_{size}(t)$	$si(t)$
0 to 1	10	2	10	T_{cycle}/δ Columns
1 to 0	10	2	10	
1 to 2	10000	2	10	
1 to 3	0	2	10	
2 to 1	10	2	10	
2 to 4	0	2	10	
3 to 1	10	2	10	
3 to 4	0	2	10	
4 to 2	10000	2	10	
4 to 3	0	2	10	

Fig. 4. Example state when flow F_2 attempts routing on switch sw_3.

- *TSLR*'s action space can be expressed easily as discrete since in involves selecting a switch.
  Therefore, we require a DQN algorithm that performs efficiently in a discrete action space and is less sensitive to model fine-tuning than other methods with an Actor–Critic manner.
- C-DQN's distributional characteristic allows for more flexibility in making assumptions and stronger inferences into learning problems.
  For this reason, C-DQN allows *TSLR* to adapt easily to and perform better in a variety of environments (topologies and flow sets) without fine-tuning model hyperparameters.
- C-DQN offers superior performance in environments with a distribution of bimodal or multimodal values [34].
  Since there may be multiple routing results for proper scheduling and load balancing, this feature aligns well with our problem.
- C-DQN reveals better performance than other DQN-based algorithms such as Double DQN, or Dueling DQN [33].

Therefore, we adopt *C-DQN* in the design of *TSLR*, and *C51 Algorithm* [21] provides an implementation of C-DQN in TensorFlow's TF-Agents [34] for our work.

4. Design

This section presents the *TSLR* design, which consists of *DRR* and *PSS*algorithms.

4.1. *DRR* — *Distributional RL-based Routing*

TSLR's goal is to discover a routing path set that satisfies the deadlines for all flows. While doing so, *DRR* balances the load on links to the extent possible to improve the network's effective capacity. Therefore, the objective function is as follows:

$$\max \sum_{t=1}^{T} (SCH(t) + L(t) + P(t)) \tag{4}$$

$SCH(t)$ is the *schedule score function* to confirm how well a route set (when scheduled upon) satisfies the flow deadlines, $L(t)$ is the *load balancing score function*, and $P(t)$ is the *punishment function* for invalid actions. t is the episode's time-step. The end-to-end routing of one flow is considered one episode. For each flow, three functions are combined as in Eq. (4) to consider all rewards related to scheduling, load balancing, and punishment. *DRR* obtains the reward value according to the sum of three functions when one episode has ended. However, to validate whether flows meet their deadlines and calculate the generated routes' reward, scheduling must be preceded for each route set. For this purpose, *TSLR* executes its scheduling algorithm *PSS* (Section 4.2).

DRR addresses the complexity problem using RL. The main challenge is to define the state, action, and reward/penalty of RL that achieves our goal.

STATE: The state matrix $s(t)$ of *DRR* is composed of three column vectors and a matrix as,

$$s(t) = \{f_{path}(t),\ f_{interval}(t),\ f_{size}(t),\ si(t)\} \tag{5}$$

where each row corresponds to a link. Fig. 4 provides an example state when flow F_2 tries to route from sw_4 in Fig. 3. In this example scenario, the assumed frame size of each flow is 125 bytes and the link bandwidth is 100 Mbps.

$f_{path}(t)$ is a vector that represents the location, destination, and current flow's path until t. $f_{path}(t)$ distinguishes links into three cases. First are the links that can be reached from the current switch, including those that directly connect toward the flow's destination. If the F_2's current location in Fig. 3 is sw_4, it can move to sw_2. In this case, as exemplified in Fig. 4, a relatively large value (e.g., 10000[2]) represents the reachable links and clearly emphasizes the sender and receiver at t. Second are the links toward the switches that are already visited by the flow, assigned a value of 0 to prevent further consideration. Finally, the rest are initialized to $f_{size}(t)$ of the corresponding link to illuminate the minimum potential free capacity on the link. We observe that the learning instability due to deviations from these values can be suppressed through a C-DQN's distributional learning strategy with mini-batch learning.

$f_{size}(t)$ depicts the packet size (per transmission) of the current flow to be scheduled, represented as the transmission time required for one packet to be scheduled within T_{cycle}[3] on each link. $f_{size}(t)$ is set to $\lceil$'$packet\ size\ per\ transmission$' $\div$ '$link\ bandwidth$'$\rceil$. This conversion represents $f_{size}(t)$ with time units equal to that of schedule information $si(t)$. $f_{size}(t)$ must be expressed explicitly on every row because the bandwidth of each link may differ (e.g., 100 Mbps, 1 Gbps, etc.).

$f_{interval}(t)$ embeds the current flow's tx interval (period) given by the *ratio of* T_{cycle} *to flow interval* in all rows.

$si(t)$ matrix represents the current scheduling information of all network links. To reduce the state matrix's size for faster runtime, scheduling information $si(t)$ is expressed using '$T_{cycle} \div \delta$' columns with an adaptive compression value δ for reducing $si(t)$ as follows:

$$\begin{gathered} si(t)_{ij} = \delta - \sum_{k=\alpha_j}^{\beta_j} Schedule^t_{(i,k)} \\ \alpha_j = \delta \cdot j,\ \beta_j = \delta \cdot (j+1) - 1 \\ 0 \le i < Number\ of\ links,\ 0 \le j < T_{cycle} \div \delta \end{gathered} \tag{6}$$

i and j represent the row (link) and the column (time), respectively. Therefore, α_j and β_j mean the beginning and end of the time range assigned to column j. All columns of $si(t)$ are initialized to δ, the size of their allocated time range. $Schedule^t_{(i,k)}$ indicates in binary that link i is scheduled at time k. For example, if link A is reserved at 0 μs, $Schedule^t_{(A,0)}$ equals, 1 but otherwise 0. Therefore, as the reservation progresses, $si(t)_{ij}$ decreases from δ to 0.

A larger δ makes smaller $si(t)$ matrix size, seeking faster learning. However, the more the $si(t)$ size is compressed by δ, the more abstract scheduling information becomes. Therefore, by adjusting the δ according to the environment in which *DRR*is to be run, it is possible to control

[2] This can be any value that is sufficiently greater than $f_{size}(t)$.
[3] In TAS, gates are scheduled within a *cycle time*, T_{cycle} and the GCL schedule is repeated every T_{cycle}. T_{cycle} should be set as the least common multiple (LCM) of the flow intervals passing through the switch (more details in Section 4.2).

the trade-off between accuracy and latency; i.e., accurate expression of scheduling information versus faster *DRR* operation. For example, if *DRR* runs in a high-performance GPU environment, δ could be set to 1 to reflect accurate scheduling information; otherwise, δ could exceed 1 to reduce the execution time.

In our scheme, we assume that each end system is integrated with a switch ('talker' and 'listener' in TSN terminology). If end systems and switches are separated, the state matrix $s(t)$'s row should contain additional links connected to the end systems. However, if an end system connects only to one switch, then that link can be excluded safely from $s(t)$ to reduce size and complexity. Therefore, in this case, the sources of flows from end systems are replaced by a directly connected switch.

ACTION: An *action* $a(t)$ is to choose a switch to visit next for routing. Therefore, if switches are indexed as an integer starting from 0, the range of $a(t)$ that the agent can take is between 0 and *'number of switches — 1'*. However, selecting a switch not directly connected to the current location would be an invalid action. Therefore, *DRR* avoids this by removing those actions from the action space; i.e., an action space is defined by links toward each neighbor of the current location. As exemplified in Figs. 3 and 4, if flow F_2 tries to route from sw_3, a valid action to take is either a link to sw_1 or to sw_4.

There are also other *invalid actions* that cannot be known in advance before routing or scheduling and thus must be handled by giving a *penalty* in the reward. Selecting a switch that generates a routing loop is an invalid action. Also, an action that forces link utilization to exceed 100% of its link bandwidth is an obviously invalid action. Finally, to achieve global load balancing, *DRR* compares the maximum link utilization of the current path set to that of previously scheduled sets.[4] Subsequently, an action that increases the max link utilization is defined as an invalid action. These *invalid actions* will be punished with a *penalty* in the reward.

REWARD: The reward function $r(t)$ is composed of,

$$r(t) = \begin{cases} SCH(t) + L(t) & if\ a\ valid\ action \\ P(t) = r_{punishment} & if\ an\ invalid\ action \end{cases} \tag{7}$$

DRR first determines whether an action is valid or invalid and assigns a reward $r(t)$ in Eq. (7) accordingly. If an action is invalid, the reward function $r(t)$ has a penalty value $P(t)$ according to $r_{punishment} = -N - 1$, which is smaller than the most negative value that a valid action can take as a reward. N is the number of switches in the topology. The reason that *DRR* assumes a value of $-N-1$ as $r_{punishment}$ is that $r_{punishment}$ must be less than the reward for any action. The worst action reward that can theoretically be received without falling into the penalty state is the minimum value of $SCH(t)+L(t)$. $SCH(t)$ is a reward according to the schedule score function in Eq. (8), and $L(t)$ is a reward according to the load balancing score function in Eq. (9). The minimum of $SCH(t)$ is 0 (when the binary value is 0) and the minimum of $L(t)$ will be $-N$ according to Eq. (9). Therefore, $r_{punishment}$ must be less than $-N$. However, if the $r_{punishment}$ becomes too small, the reward range becomes too large and the value distribution precision of C-DQN decreases. Therefore, $r_{punishment}$ possesses the value of $-N-1$ to narrow the reward range.

DRR's primary goal involves creating a routing path set that satisfies the deadlines of as many flows as possible when scheduled. For this purpose, $SCH(t)$ reflects whether the scheduling result complies with the deadlines of flows; i.e., the more flows comply with the deadline, the greater the reward;

$$\begin{aligned} &SCH(t) = b_{dst} \cdot b_{fdl} \cdot (N + N \cdot (F \cdot F)^{ComplianceRate(t)-1}) \\ &\quad b_{dst}\ :\ Has\ the\ flow\ arrived\ at\ the\ destination? \\ &\quad b_{fdl}\ :\ Does\ the\ flow\ latency\ satisfy\ its\ deadline? \end{aligned} \tag{8}$$

[4] Only for those that have successfully scheduled all flows.

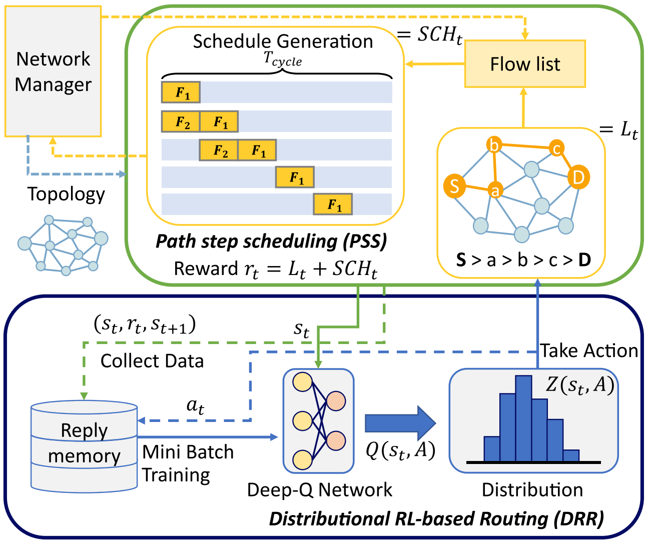


Fig. 5. TSN scheduler with RL-based routing (*TSLR*).

b_{dst} and b_{fdl} are binary values that are 1 if each condition is satisfied and 0 otherwise. N is the number of switches in the topology. and F is the number of flows. *ComplianceRate(t)* represents the ratio of how many flows succeeded in meeting their deadlines among the total flow set. *DRR* accepts a routing result when it does not decrease *ComplianceRate(t)*. $SCH(t)$ carries value N by default to provide a reward for successful routing of one flow. Also, $SCH(t)$ awards an additional reward as a function of the overall scheduling result with *ComplianceRate(t)*. However, in order for the reward to have a meaningful distribution, its value must be clearly distinguished according to the scheduling result. The simplest method involves setting an extremely large change in the reward's absolute value according to the scheduling result, but this makes other rewards meaningless. Therefore, *DRR* takes $F \cdot F$ as the base of the exponential function in the reward equation for the scheduling result. A higher *ComplianceRate(t)* produces greater a change in the reward. Finally, N is multiplied to increase the maximum value according to N.

DRR has two policies for load balancing, global and local. The global load balancing policy employs *min–max fairness* criteria to achieve "minimizing the maximum link utilization" using the penalty function for invalid actions. On the other hand, the local load balancing policy is added as $L(t)$ under the intuition that avoiding links with higher utilization is helpful for future scheduling, even in the local scope. Note that if $L(t)$ is made positive, the reward may increase as the path length increases, which may incorrectly encourage *DRR* to select unnecessarily longer paths. To prevent this, $L(t)$ is always a negative value as follows:

$$L(t) = -N \cdot 1024^{LinkUtilization(t)-1} \tag{9}$$

$LinkUtilization(t)$ refers to the link utilization of the current link. When link utilization increases to 10% (0.1), the agent receives a worse reward by a factor of two ($1024^{0.1} = 2$). The reason for multiplying by N is related to the reward's range. According to Eq. (7) and Eq. (8), the range of rewards increases as N increases. However, if $L(t)$ has a fixed range, it will hardly affect the overall reward value. Thus its influence on the value distribution can disappear. Therefore, N is multiplied to increase $L(T)$'s influence on a reward function.

Fig. 5 illustrates an overview of *TSLR*. When a flow list on a network topology is given as the input, and their paths are initialized to the shortest paths, *DRR* iteratively finds alternate routes for each flow and replaces them according to its policy. *DRR* learns for a certain period by evaluating alternate routes' rewards in terms of TAS scheduling

Algorithm 1 Path Step Scheduling (*PSS*)

```
Input: flowList including the routing paths generated by the DRR, and $T_{cycle}$
1: scheduleSet ← {}
2: $T_{cycle}$ ← get$T_{cycle}$(flowList)
3: maxLength ← maxPathLength(flowList)
4: flowsGroups ← groupedByInterval(flowList)
5: sortedGroups ← sortedByInterval(flowsGroups)
6: for group in sortedGroups do
7:     itrNum ← INT($T_{cycle}$ ÷ group.interval)
8:     for itr in range(itrNum) do
9:         for step in range(maxLength) do
10:            curFlows ← {}
11:            for flow in group do
12:                if step ≤ flow.pathLength then
13:                    link ← getLink(flow, step)
14:                    push(curFlows[link], flow)
15:                end if
16:            end for
17:            for link in curFlows do
18:                flows ← sortedByDeadline(curFlows, link, itr)
19:                for f in flows do
20:                    slots ← getTimeSlots(scheduleSet, link)
21:                    scheduleSet ← setSchedule(f, slots, scheduleSet, link, itr)
22:                end for
23:            end for
24:        end for
25:    end for
26: end for
```

success rate and load balancing according to Eq. (7). *DRR* training and evaluation repeat continuously, and the actual routes are reflected gradually during this process. Because the problem's form could vary significantly according to the network topology or flow set, *DRR* trains without a pre-trained neural network in other networking scenarios.

4.2. PSS — Path Step Scheduling

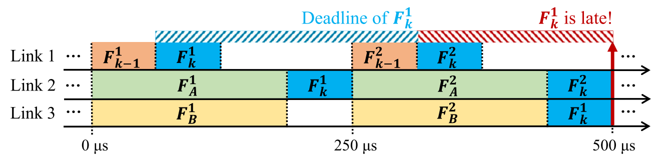
The *PSS* algorithm receives the routing result generated by *DRR*as an input and generates the network's schedule. Algorithm 1 is the pseudo-code of *PSS*. We use the example scenario in Fig. 3 to describe *PSS* operation and assume that flow F_0 has a 500 μs interval and F_1 and F_2 have 250 μs intervals. The flow deadlines of flows match their interval.

Initialization (Line 1–5): *scheduleSet* initializes as a set of schedule lists (i.e., GCLs repeated with T_{cycle}) for each switch in the network. T_{cycle} is calculated as the least common multiple of the intervals of flows in *flowList*, and *maxLength* is set as the maximum path length (hop-count) of all flows. Finally, the flows with the same interval are grouped together, and these groups are sorted in ascending order of the intervals. For the example in Fig. 3, initialization occurs as follows:

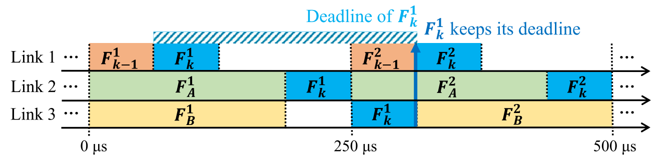
$$\begin{aligned} &scheduleSet = \{\},\ T_{cycle} = 500\ \mu\text{s},\ maxLength = 2, \\ &sortedGroups = \{[F_1, F_2], [F_0]\} \end{aligned} \tag{10}$$

Line 6∼8: Scheduling is performed in group order by smaller intervals, proceeding as many times as the number of transmissions (*itrNUM*) within one T_{cycle} (ith *itr*). For example, if T_{cycle} is 500 μs and a flow's interval is 250 μs, that flow will transmit twice within T_{cycle} (reve.g., *itrNUM* = 2 in line 8). Furthermore, not all flow transmissions are scheduled at once, but ith transmissions of the same flow group are scheduled in a batch. Therefore, in the example of Fig. 3, the scheduling proceeds in the order of (1) the first transmissions of F_1 and F_2, (2) their second transmission, and (3) the transmission of F_0.

The reason for scheduling flows with smaller intervals first is that their constraints (deadlines) are tighter, and when multiple transmissions are performed within T_{cycle}, ith transmissions are affected by the



(a) Schedule all transmissions of a flow at once.



(b) Schedule itr^{th} transmissions of a flow at once.

Fig. 6. Difference in order of scheduling transmissions.

time occupied by previous transmissions. Furthermore, the reason not to schedule all transmissions of one flow simultaneously is to ensure fairness to other flows. For example, consider Fig. 6(a), where T_{cycle} is 500 μs, and the intervals and deadlines of flows F_{k-1}, F_k, and groups of flows F_A and F_B are all 250 μs. When flow F_k utilizes links in the order of 1-2-3, flow F_k is unable to comply with the deadline (i.e., F_k^1 is not delivered within 250 μs) due to the second transmission of F_B (F_B^2). This shows that scheduling all interval transmissions of one flow at once imposes a bigger constraint on the next flow scheduling. Therefore, *PSS* schedules all ith interval transmissions within the same flow group first and then schedules (i+1)th interval transmissions. This method, for example, enables flow F_k to comply with its deadline by scheduling F_k^1 earlier than F_B^2 on link 3, as shown in Fig. 6(b).

Line 9: As the name suggests, *PSS* schedules step-by-step, where a step indicates a path hop. After scheduling the first hop of all flows in one iteration, scheduling the second hop for each flow occurs. In Fig. 3, $\{F_1 : 0 \to 1\}$ and $\{F_2 : 3 \to 4\}$ are scheduled first, and then the scheduling of $\{F_1 : 1 \to 2\}$ and $\{F_2 : 4 \to 2\}$ follows.

Line 10∼17: This part confirms which flow should be scheduled on which link in the current step. The algorithm identifies the mth-step (mth hop) link of the flow path and stores the flow that should be reserved in the corresponding link. When this process is completed, the flow-link lists that need to be scheduled at the current step are stored in *curFlows*.

Line 18∼22: *PSS* reads the list of flows and links that should be scheduled and performs actual scheduling. The list of flows is sorted in ascending order of the remaining deadline per remaining hop so that a flow with an imminent deadline is scheduled first. Then, *getTimeSlots* verifies the free space (called *timeslot*) within T_{cycle} that can be scheduled on the current link. In *setSchedule*, a flow is scheduled in a suitable timeslot with a consideration of its constraint. If there are multiple timeslots, the algorithm confirms whether sufficient space exists to be scheduled and to satisfy the flow's constraints sequentially from the front. In case of that there is a favorable determination, the scheduler selects the corresponding timeslot. Otherwise, the scheduler checks the next timeslot, and if there are no available timeslots, it stops scheduling.

getTimeSlots() in line 21: In most TAS scheduling, a timeslot can have a maximum length of T_{cycle}, as shown in Fig. 7. In general, a timeslot having a length of T_{cycle} is divided into several sub-timeslots in advance, and then a schedule is assigned to these sub-timeslots, or timeslots are split into two by dividing the original timeslot centered at the scheduled part. In both approaches, it is assumed that a GCL schedule cannot extend beyond the endpoint of the original timeslot of length T_{cycle}. *PSS* is similar to the latter approach. However, *getTimeSlots()* of

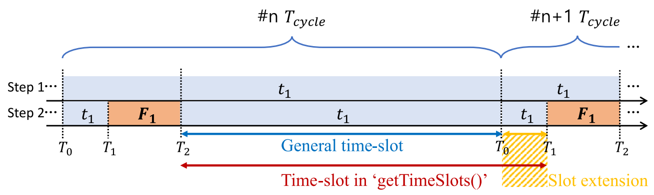


Fig. 7. Example of *getTimeSlots()*.

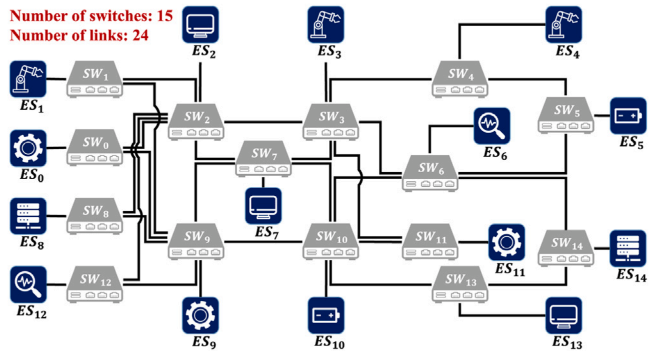


Fig. 8. Orion CEV network topology [35–37].

PSS exploits the fact that the TAS schedule is repeated. For example, a new timeslot at step-2 in Fig. 7 ends at T_1 beyond T_0, the endpoint of T_{cycle}. This allows *PSS* to expand the scheduling space, increasing schedulability.

Computational complexity of *PSS* depends mainly on the number of operations scheduling links. There are additional operations, such as flow sorting, but they are relatively insignificant. The number of link scheduling operations N_{link} can be expressed as:

$$N_{link} = \sum_{\forall f \in F} Length(f_{path}) \cdot (T_{cycle} \div f_{interval}) \tag{11}$$

The number of links to be scheduled per-flow depends on the number of links used in the flow path ($Length(f_{path})$) and how many times to schedule each link according to the flow's transmission interval ($T_{cycle} \div f_{interval}$). Since scheduling must be performed '$T_{cycle} \div f_{interval}$' times for each link, the number of link scheduling required by one flow is $Length(f_{path}) \cdot (T_{cycle} \div f_{interval})$. If this is aggregated for all flows f in the entire flow set F, it is the total number of link scheduling (N_{link}) performed by *PSS*. Therefore, *PSS* has $O(N_{link})$ complexity, which depends on the transmission interval and flow path length.

5. Evaluation

We evaluate *TSLR* against two state-of-the-art algorithms, *Joint Routing and Scheduling (JRaS)* [16] and *ILP-Red+Conf+Adv (IRCA)* [20], and also against *Simulated Annealing*-based dynamic routing with *PSS (SA/PSS)* for comparison with a heuristic approach. *SA/PSS* is a metaheuristic algorithm that finds a better solution by changing the current route to other routes from the entire set while applying the same *PSS* for scheduling. The main point of comparison with *SA/PSS* is whether *DRR*'s RL-based routing proves effective.

We conduct an extensive set of simulations using multiple different network topologies to evaluate *TSLR*'s scalability. We compare the scheduling time, schedulability, scheduled flow latency and jitter, maximum link utilization, and path length distribution.

Table 2
Random topology specifications for the simulation.

Topology	Number of switches	Number of links	Number of flows
T10	10	20	100
T20	20	40	200
T30	30	60	300
T40	40	80	400
T50	50	100	500

Table 3
Traffic specifications for the simulation.

Traffic name	Traffic size	TX interval	Latency/deadline	N flows
ST-1	128 bytes	600 μs	100 μs	$N/5$
ST-2	96 bytes	400 μs	100 μs	$N/5$
ST-3	96 bytes	300 μs	100 μs	$N/5$
ST-4	64 bytes	200 μs	100 μs	$N/5$
ST-5	64 bytes	100 μs	100 μs	$N/5$

5.1. Simulation setup

We implement *TSLR* on Tensorflow's Agents [34], *JRaS* and *IRCA*[5] on Gurobi(v9.5),[6] and *SA/PSS* on Python(v3.9).[7] Simulations are conducted on a workstation with an Intel Core i7-10700F, 32 GB RAM, 96 GB virtual RAM(SSD), and no GPU.

We use ten different network topologies for evaluation. The first is NASA's Orion Crew Exploration Vehicle (CEV) network topology in Fig. 8 [35–37] which consists of 15 switches. The five topologies in Fig. 9 are randomly generated according to the specifications in Table 2 to reveal that our scheme can generalize to non-specific topologies. For these random topologies, we use $2n$ links for n nodes. This is a popular approach to support two-link redundancy using industrial fault-tolerance protocols such as *high-availability seamless redundancy* (HSR) and *parallel redundancy protocol* (PRP) specified in IEC 62439-3, or TSN's *frame replication and elimination for reliability* (FRER) in IEEE 802.1CB [38]. In addition, to observe the impact of nodes-links ratio on *TSLR*, four additional random topologies having the same number of nodes or that of links with the T30 graph (Fig. 9(c)) as used as in Fig. 11. All links are full-duplex with 100 Mbps bandwidth, and all schemes use up to 75% of link bandwidth for scheduling to comply with the TSN standard.

Five ST flow types are used as described in Table 3, and we vary the total number of flows from 40 to 200 in the CEV scenario, and 100 to 500 in random topology scenarios; one fifth of the flows are of each type. For the topologies in Fig. 11, same number of flows as the T30 scenario (300 flows) is used. We set the flow deadlines to 100 μs according to industry standards [39,40] and use five different flow intervals to simulate a complex IIoT scenario. *DRR* parameter settings are listed in Table 4. In the case of δ, we adopt 5 as a default value for the CEV scenarios. However, in random topology scenarios, δ is adjusted to reduce the size of *TSLR*'s state matrix. In T50, δ is 20, and in other random topology scenarios, δ is 10.

5.2. Performance of routing and scheduling

Scheduling time is defined as the earliest time that the schedules for all flows are created while satisfying the deadline and bandwidth requirements. Scheduling time is very important for TSN since it directly impacts the network's schedulability within a given time limit. Since

[5] The original *IRCA* algorithm [20] does not support scheduling heterogeneous flows having different transmission intervals, and therefore, we had to enhance it to support the scheduling of flows with heterogeneous intervals.
[6] Gurobi Optimization, https://www.gurobi.com.
[7] https://www.python.org/downloads/release/python-390/

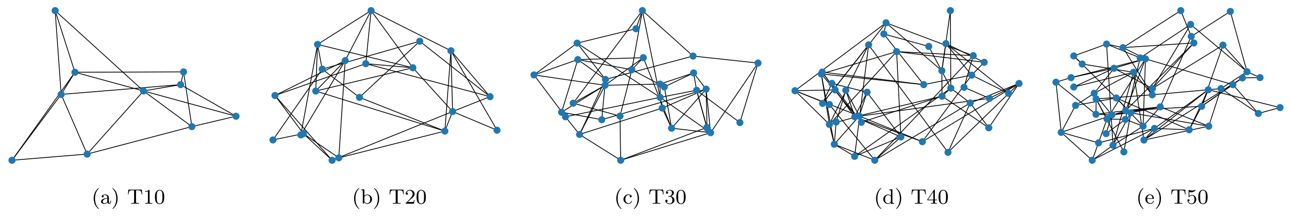


Fig. 9. Random network topologies generated based on Table 2.

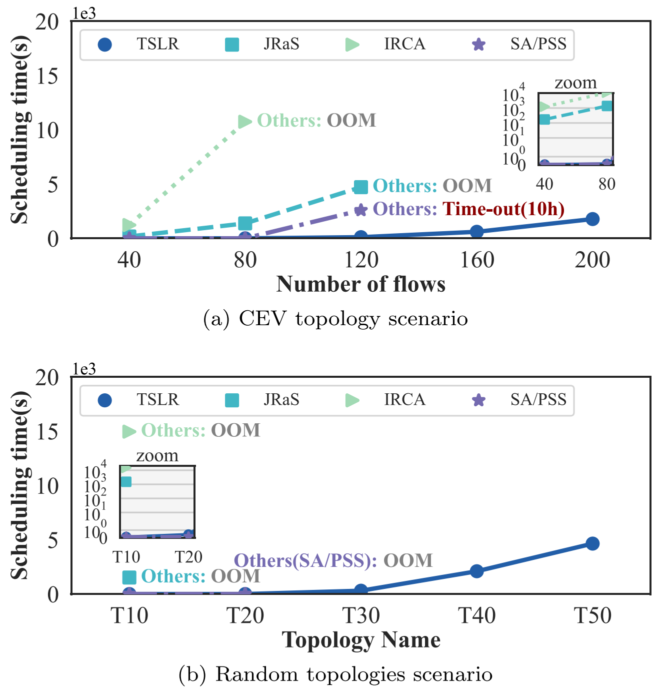


Fig. 10. Comparison of scheduling time for the four algorithms. (Missing data point means scheduling failed to complete. Only *TSLR* succeeds beyond 160 flows or T30 scenario respectively.)

Table 4
Settings of *TSLR*.

Name	Value
Neural network	FC (100, 100, 100)
Number of atoms	51
Optimizer	Adam
Activation function	Leaky ReLU
Learning rate	0.0001
Minibatch size	32
Discount rate	0.999
Loss function	Mean Squared Error
Replay buffer size	100000
Max Q value	2·(Number of Nodes)
Min Q value	–(Number of Nodes)–1
Default δ	5

TSLR is not a pre-trained model, the *scheduling time* of *TSLR* includes both training time and evaluation (testing) time. Note that *scheduling time* result exists if and only if the scheduling succeeds. In other words, there is no *scheduling time* result if scheduling fails. Thus, *scheduling time* result also shows the *schedulability*.

The *scheduling time* is influenced not only by the size and bandwidth of the network topology but also by the number, size, and requirements of the flows, i.e., traffic load. Fig. 10(a) plots the *scheduling time* as we increase the traffic load on the CEV topology. *TSLR* is the fastest for all cases, and the execution time of other algorithms increases dramatically as the number of flows increases. For example, in a simulation with 120 flows, *TSLR* shows a scheduling time of about 3605 s faster than *JRaS* and 2478 s faster than *SA/PSS*. In addition, *JRaS* fails to schedule 160 flows and beyond due to an out-of-memory (OOM)[8] error, and *IRCA* is unable to find a schedule beyond 80 flows. On the other hand, *TSLR* schedules about 66% more flows than *JRaS* and 150% more flows than *IRCA* while successfully scheduling 200 flows, showing significantly improved schedulability.

These results of *JRaS* and *IRCA* are worse than what has been presented in [20] because *IRCA* is designed for flow sets with the same intervals. According to these results, the computing resources required by the ILP-based approaches increase rapidly according to the number of flows to be scheduled (with multiple transmissions due to interval and complicated hyper-period), and the chances of finding a solution drop rapidly. *SA/PSS* had similar results with the ILP-based approaches as well. Although OOM does not occur, *SA/PSS* cannot find solutions within ten hours in 160 flow cases and beyond. These results illustrate that the ILP-based methods and the heuristic method of changing the path from the entire path set are inefficient. *TSLR* succeeds in generating schedules for a complex interval flow set in a relatively short time in all cases, showing significantly improved schedulability and scheduling time. This confirms the efficacy of an RL-based routing method like *DRR*.

Fig. 10(b) plots the *scheduling time* results for random topologies as we increase the network size and the number of flows. In order to maintain a similar proportional traffic load, both the network size and the number of flows are increased at the same rate as in Table 2. It can be seen that the increase in network size (which increases the average path length of flows) and the number of flows have a significant impact on scheduling complexity and, thus, the *scheduling time*. Nevertheless, *TSLR* still significantly outperforms the ILP-based schemes. *JRaS* and *IRCA* consume substantial time on T10 and returns out-of-memory on larger scenarios. *SA/PSS* succeeded in scheduling very quickly in T10 and T20. However, this is because *PSS* successfully schedules based on the initial routing result without executing the SA-based routing algorithm. In the other scenarios, *SA/PSS* returns OOM. This shows that even generating a routing set is difficult in a scenario with a relatively large topology, and that *TSLR* is more effective in terms of both schedulability and scheduling time than other approaches for different topologies and flow sets. *TSLR* succeeds in routing and scheduling input flows in various topologies without a pre-trained model because RL-based routing of *DRR* can find reasonable routes even when unable to view the entire routing set.

Finally, to understand the impact of node–link ratio on *TSLR*, we used T30 topology as a reference and varied the number of nodes and links. Specifically, we used a fixed number of nodes from T30 and different number of links, and vice versa, as shown in Fig. 11. Fig. 12 plots the scheduling result on these topologies. The bar graph represents the number of flows whose schedules meet their deadline requirements, and the line graph represents the scheduling time for the best result within a time limit (10 h, red dotted link). In general, when the number of links increases or the number of nodes decreases, there

[8] We had 128 GB of RAM (32 GB physical + 96 GB virtual) on our simulation machine.

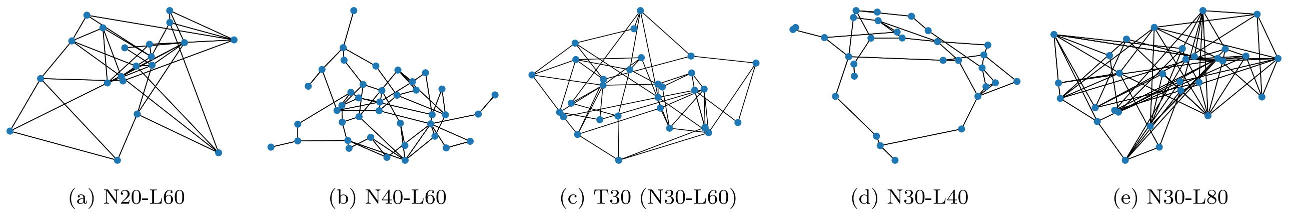


Fig. 11. Fixed random network topologies with the fixed number of nodes or links. In each subfigure's name, the number after N means the number of nodes, and the number after L means the number of links.

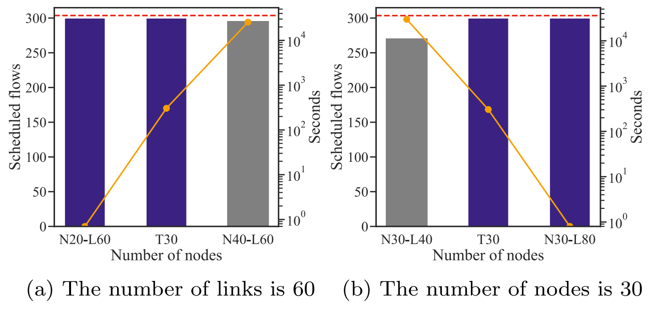
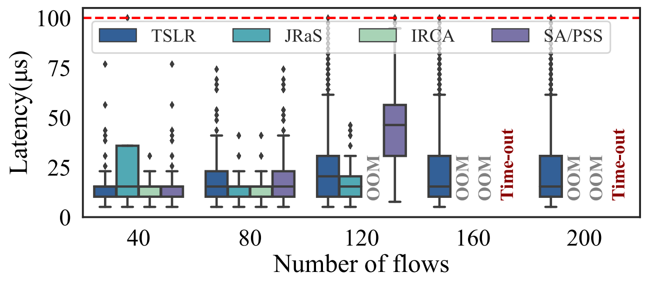


Fig. 12. Scheduling result of *TSLR* in fixed random graphs. The grey bars mean that *TSLR* fails to configure successful TAS schedules in 300 flows.

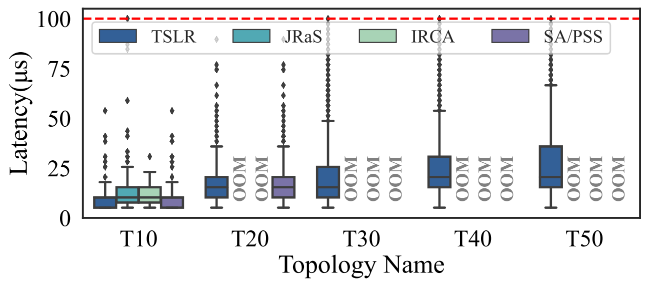
are abundant links such that shortest path is sufficient for scheduling without the need for alternative routes search by RL. On the other hand, when the number of nodes increases or the number of links decreases, there is an insufficient number of links to fully support (schedule) the given set of flows. Specifically, in the N40-L60 scenario (Fig. 11(b)) with 40 nodes and 60 links, only 295 flows were scheduled successfully within the time limit, and only 270 flows succeeded in the N30-L40 topology (Fig. 11(d)). This shows that the node–link ratio of the network greatly affects the scheduling performance given a fixed number of flows; if there are too many links, alternate routes are not necessary, and if there are too few links, the number of flows that can be scheduled must be reduced accordingly.

Overall, the evaluation results show that it is impractical to have all paths as a search space, but alternate paths should be explored to increase the schedulability of TAS, and *TSLR* addresses this problem appropriately. *TSLR* succeeds in routing and TAS scheduling with significantly less running time and memory usage than existing methods in all evaluated scenarios.

Latency/Jitter: Fig. 13 plots the latency of ST flows, showing the latency results when the *scheduling time* was measured. In other words, when the first successful schedule was created. The red dotted horizontal line is the deadline that the flows must satisfy. Since we define *successful scheduling* as generating a schedule that satisfies the latency requirements for all time-critical flows, there is no flow that violates the deadline. Whether such a schedule can be found was already reflected in the aforementioned *scheduling time* or *schedulability* results. Nevertheless, *TSLR*'s overall average latency is similar to *JRaS* and *IRCA* and quite lower than *SA/PSS*. For example, the average latency of *TSLR* in all CEV scenarios is below 25 μs, which is 25% of the flow's latency requirement. (The reason *TSLR* and *SA/PSS* have the same latency in most smaller scenarios is that *PSS* successfully scheduled on the initial routing without needing to explore alternatives.) This implies that *TSLR* effectively generates a schedule considering deadlines. The optimization functions of *JRaS* and *IRCA* are to reduce latency. In contrast, *TSLR* considers a detour path (maybe longer) for load balancing. However, in the T10 scenario, *TSLR*'s latency is lower than that of *JRaS* and *IRCA* and markedly lower than that of *SA/PSS* in the CEV 120 flows scenario. Finally, Tables 5 and 6 show that *TSLR* maintains acceptable



(a) Latency of ST-1 in CEV.



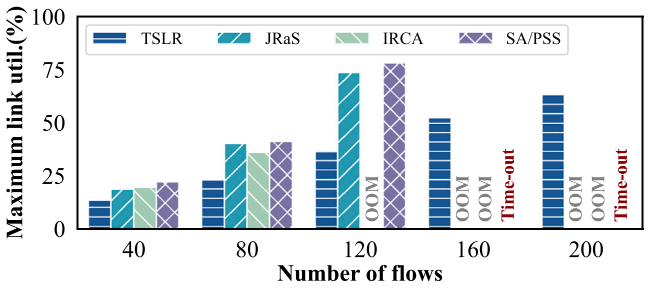
(b) Latency of flows in Table 2

Fig. 13. Latency results of flows for simulations of routing and scheduling Missing results mean scheduling failed.

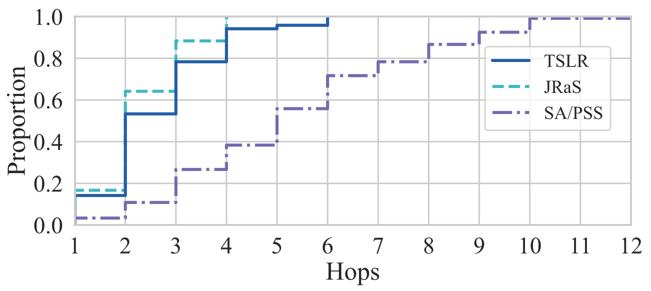
average jitter, being within 5μs in most scenarios. Although this is not as good as *JRaS* and *IRCA* that achieve zero jitter, their schedulabilities are poor. The main results is that *TSLR* is sacrificing a little latency and acceptable jitter for significantly improved schedulability. Therefore, we conclude that *TSLR* is competitive when considering schedulability, latency, and jitter jointly.

Load balancing and path length (Hops): Fig. 14(a) displays the load balancing performance of *TSLR*. Maximum utilization is the utilization of the bandwidth allocated to TSN. According to Fig. 10(a), *TSLR* generates successful schedules in less than an hour on average for all CEV scenarios. However, in this simulation, we fixed the execution time of *TSLR* to one hour to obtain improved load balancing results. In all scenarios of Fig. 14(a), *TSLR* achieves the smallest maximum utilization compared to all other methods. Particularly, in the CEV-120, the maximum utilization of *TSLR* is approximately 50% lower than that of *SA/PSS*. This indicates that the flows have been distributed to alternate links for balanced traffic load, and there is more available bandwidth for other later flows, including best-effort traffic.

Fig. 14(b) plots the empirical cumulative distribution function (eCDF) of flow path lengths for CEV 120 flows scenarios in Fig. 14(a). *IRCA* is excluded because scheduling failed in the CEV 120 flows scenario. Since *TSLR* considers detour routes (from shortest paths) for load balancing, path length increases are inevitable. However, compared to *JRaS*, the increases are small and insignificant, considering the benefits gained. Additionally, compared to *SA/PSS*, the overall



(a) Maximum link utilization



(b) Hop distribution

Fig. 14. Maximum link utilization and hop distribution in CEV.

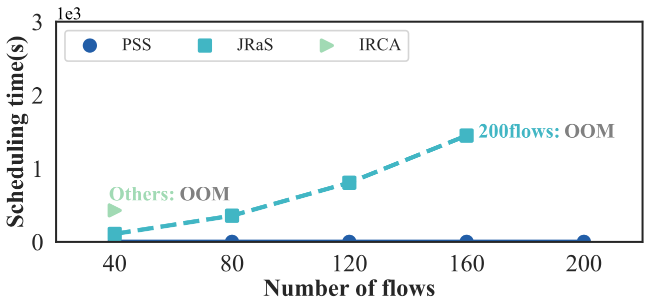


Fig. 15. Scheduling time with static route on CEV topology.

Table 5
Average jitter in CEV scenarios. '–' means no result can be obtained because it failed to schedule all flows.

Number of flows	*TSLR*	JRaS	IRCA	SA/PSS
40	3.285 μs	0 μs	0 μs	3.285 μs
80	1.829 μs	0 μs	0 μs	1.829 μs
120	4.617 μs	0 μs	–	7.890 μs
160	4.894 μs	–	–	–
200	5.601 μs	–	–	–

Table 6
Average jitter in random topology scenarios. '–' means scheduling failed.

Number of flows	*TSLR*	JRaS	IRCA	SA/PSS
40	0.162 μs	0 μs	0 μs	0.162 μs
80	2.334 μs	–	–	2.334 μs
120	3.520 μs	–	–	–
160	4.656 μs	–	–	–
200	4.907 μs	–	–	–

lengths of the *TSLR*-generated paths are notably shorter. This means that *TSLR*'s negative reward policy for routing effectively suppresses the increase in path lengths while pursuing the schedulability and load balancing goals.

Table 7
The routing result in CEV 40 scenario.

Flow ID	Flow size	Flow interval	Path
0	128 bytes	600 μs	3 → 7
1	128 bytes	600 μs	8 → 2 → 3
2	128 bytes	600 μs	12 → 9 → 8
3	128 bytes	600 μs	10 → 13
4	128 bytes	600 μs	12 → 9 → 8
5	128 bytes	600 μs	10 → 9 → 8
6	128 bytes	600 μs	2 → 3 → 6
7	128 bytes	600 μs	1 → 9
8	96 bytes	400 μs	6 → 10 → 9 → 12
9	96 bytes	400 μs	13 → 14 → 6 → 5
10	96 bytes	400 μs	4 → 5 → 6 → 3
11	96 bytes	400 μs	7 → 10 → 13
12	96 bytes	400 μs	2 → 7 → 9 → 10
13	96 bytes	400 μs	6 → 10 → 13
14	96 bytes	400 μs	5 → 4 → 3 → 7 → 2 → 12
15	96 bytes	400 μs	14 → 13 → 10 → 9 → 8
16	96 bytes	300 μs	7 → 3 → 6
17	96 bytes	300 μs	1 → 9
18	96 bytes	300 μs	11 → 3 → 4
19	96 bytes	300 μs	10 → 6 → 5
20	96 bytes	300 μs	5 → 6 → 3
21	96 bytes	300 μs	1 → 2
22	96 bytes	300 μs	14 → 6 → 3
23	96 bytes	300 μs	0 → 2 → 1
24	64 bytes	200 μs	11 → 10 → 6 → 14
25	64 bytes	200 μs	9 → 1
26	64 bytes	200 μs	14 → 6 → 3 → 3 → 2 → 7 → 9
27	64 bytes	200 μs	5 → 4 → 3 → 7 → 2 → 12
28	64 bytes	200 μs	11 → 3 → 6
29	64 bytes	200 μs	10 → 9 → 12
30	64 bytes	200 μs	7 → 3 → 6
31	64 bytes	200 μs	13 → 14 → 6 → 10
32	64 bytes	100 μs	14 → 13 → 10 → 7
33	64 bytes	100 μs	3 → 2 → 12
34	64 bytes	100 μs	7 → 10
35	64 bytes	100 μs	1 → 9 → 12
36	64 bytes	100 μs	1 → 2 → 8 → 9
37	64 bytes	100 μs	11 → 10 → 7 → 2 → 0
38	64 bytes	100 μs	9 → 7 → 3
39	64 bytes	100 μs	2 → 3 → 4 → 5

Table 8
Average jitter with static routing in CEV.

Number of flows	*TSLR*	JRaS	IRCA
40	3.147 μs	0 μs	0 μs
80	1.019 μs	0 μs	–
120	3.208 μs	0 μs	–
160	4.067 μs	0 μs	–
200	5.789 μs	–	–

5.3. Performance of PSS scheduler with static routing

To understand the performance of the *PSS* scheduler, we isolate the scheduling parts of the three schemes and compare them on a fixed route set generated by *DRR*, as in Table 7. Fig. 15 plots the *scheduling time* of each algorithm on the CEV topology, demonstrating that *PSS* completes TAS scheduling in less than 1 s, even for 200 flows. This is because *PSS* derives only one result for each input according to the criteria determined by the greedy algorithm. On the other hand, scheduling algorithms of *JRaS* and *IRCA* take a considerable amount of time, and out-of-memory errors occur as the number of flows increases. Due to the characteristic of *TSLR*, which recognizes a set of TAS schedules as an environment for RL, a single scheduling execution on a route set produced by *DRR* must complete quickly. Thus, it is not feasible for *TSLR* to adopt the scheduling algorithms of *JRaS* or *IRCA* for its purpose. Furthermore, despite being a greedy algorithm, *PSS* has comparable or even superior flow latencies than the other two on the same route set, as portrayed in Fig. 16. Jitter results are similar to those with dynamic routing (See Table 8).

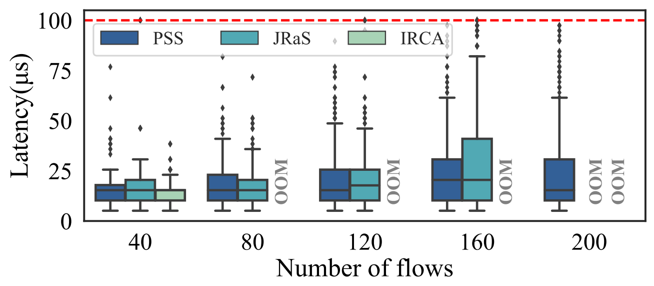


Fig. 16. Latency results of flows with static route in CEV.

6. Conclusion

In this paper, we proposed *TSLR* for TSN. It explores alternative routes for improved TAS scheduling and addresses the problem's complexity using RL. As such, it accounts for both the network load balancing and the flows' deadline to select the good scheduling option depending on the input flow set. We evaluated *TSLR* on various topologies with a diverse set of flows and compared it against two state-of-the-art algorithms, *JRaS* and *IRCA*, and also with one heuristic algorithm *SA/PSS*, to show that the scheduling performance improves while achieving *'min–max fair'* load balancing and negligible increases in path length.

However, our approach has a couple of limitations. First, when the network size grows, the learning performance deteriorates. Second, *TSLR* must be trained each time for each network scenario. We plan to investigate solutions for these challenges in our future work. Nonetheless, we believe that this work provides a reference for studies that attempt to graft machine learning to TSN. As our another future work, we plan to explore multi-path sub-stream routing in IEEE 802.1CB *frame replication and elimination for reliability* for robust TSN.

CRediT authorship contribution statement

Junhong Min: Methodology, Software, Formal analysis, Validation. **Yongjun Kim:** Conceptualization, Software, Writing – original draft. **Moonbeom Kim:** Visualization, Data curation, Investigation. **Jeongyeup Paek:** Resources, Supervision, Project administration. **Ramesh Govindan:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors are unable or have chosen not to specify which data has been used.

References

[28] IEEE Standard for Local and Metropolitan Area Network –Bridges and Bridged Networks, IEEE Std 802.1Q-2018, 1–1993, IEEE, Piscataway, NJ, 2018, http://dx.doi.org/10.1109/IEEESTD.2018.8403927, (Revision of IEEE Std 802.1Q-2014).
[1] IEEE time-sensitive networking task group, 2017, URL http://www.ieee802.org/1/pages/tsn.html. (Accessed: May 2023).
[2] Y. Seol, D. Hyeon, J. Min, M. Kim, J. Paek, Timely survey of time-sensitive networking: Past and future directions, IEEE Access 9 (2021) 142506–142527.
[3] IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic, IEEE Std 802.1Qbv-2015, IEEE, Piscataway, NJ, 2016, http://dx.doi.org/10.1109/IEEESTD.2016.8613095, (Amendment to IEEE Std 802.1Q-2014).
[4] C. Simon, M. Maliosz, M. Mate, Design aspects of low-latency services with time-sensitive networking, IEEE Commun. Stand. Mag. 2 (2) (2018) 48–54.
[5] W. Steiner, S.S. Craciunas, R.S. Oliver, Traffic planning for time-sensitive communication, IEEE Commun. Stand. Mag. 2 (2) (2018) 42–47.
[6] L.L. Bello, W. Steiner, A perspective on IEEE time-sensitive networking for industrial communication and automation systems, Proc. IEEE 107 (6) (2019) 1094–1120.
[7] A.A. Atallah, G.B. Hamad, O.A. Mohamed, Routing and scheduling of time-triggered traffic in time-sensitive networks, IEEE Trans. Ind. Inform. 16 (7) (2020) 4525–4534.
[8] S.S. Craciunas, R.S. Oliver, W. Steiner, Formal scheduling constraints for time-sensitive networks, 2017, arXiv:1712.02246.
[9] R. Dobrin, N. Desai, S. Punnekkat, On fault-tolerant scheduling of time sensitive networks, in: International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, CERTS, 2019.
[10] X. Jin, C. Xia, N. Guan, C. Xu, D. Li, Y. Yin, P. Zeng, Real-time scheduling of massive data in time sensitive networks with a limited number of schedule entries, IEEE Access 8 (2020) 6751–6767.
[11] F. Ansah, M.A. Abid, H. de Meer, Schedulability analysis and GCL computation for time-sensitive networks, in: IEEE International Conference on Industrial Informatics, INDIN, 2019.
[12] F. Dürr, N.G. Nayak, No-wait packet scheduling for IEEE time-sensitive networks (TSN), in: Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS, 2016.
[13] M. Kim, D. Hyeon, J. Paek, eTAS: Enhanced time-aware shaper for supporting non-isochronous emergency traffic in time-sensitive networks, IEEE Internet Things J. 9 (13) (2021) 10480–10491.
[14] Z. Zhou, J. Lee, M.S. Berger, S. Park, Y. Yan, Simulating TSN traffic scheduling and shaping for future automotive ethernet, J. Commun. Netw. 23 (1) (2021) 53–62.
[15] N.G. Nayak, F. Dürr, K. Rothermel, Incremental flow scheduling and routing in time-sensitive software-defined networks, IEEE Trans. Ind. Inform. 14 (5) (2017) 2066–2075.
[16] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, G. Mühl, ILP-based joint routing and scheduling for time-triggered networks, in: Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS, 2017.
[17] F. Smirnov, M. Glaß, F. Reimann, J. Teich, Optimizing message routing and scheduling in automotive mixed-criticality time-triggered networks, in: ACM/IEEE Design Automation Conference, DAC, 2017.
[18] L. Xu, Q. Xu, Y. Zhang, J. Zhang, C. Chen, Co-design approach of scheduling and routing in time sensitive networking, in: IEEE Conference on Industrial Cyberphysical Systems, ICPS, 2020.
[19] A. Alnajim, S. Salehi, C.-C. Shen, Incremental path-selection and scheduling for time-sensitive networks, in: IEEE Global Communications Conference, GLOBECOM, 2019.
[20] D. Hellmanns, L. Haug, M. Hildebrand, F. Dürr, S. Kehrer, R. Hummen, How to optimize joint routing and scheduling models for TSN using integer linear programming, in: Proceedings of the 29th International Conference on Real-Time Networks and Systems, RTNS, 2021.
[21] M.G. Bellemare, W. Dabney, R. Munos, A distributional perspective on reinforcement learning, in: International Conference on Machine Learning, 2017.
[22] K. Gong, D. Yang, W. Zhang, J. Ren, An efficient scheduling approach for multi-level industrial chain flows in time-sensitive networking, Comput. Netw. 221 (2023) 109516.
[23] N.G. Nayak, F. Dürr, K. Rothermel, Time-Sensitive Software-Defined Network (TSSDN) for real-time applications, in: International Conference on Real-Time Networks and Systems, 2016.
[24] V. Balasubramanian, M. Aloqaily, M. Reisslein, An SDN architecture for time sensitive industrial IoT, Comput. Netw. 186 (2021) 107739.
[25] C. Gärtner, A. Rizk, B. Koldehofe, R. Guillaume, R. Kundel, R. Steinmetz, Fast incremental reconfiguration of dynamic time-sensitive networks at runtime, Comput. Netw. 224 (2023) 109606.
[26] L. Yang, Y. Wei, F.R. Yu, Z. Han, Joint routing and scheduling optimization in time-sensitive networks using graph-convolutional-network-based deep reinforcement learning, IEEE Internet Things J. 9 (23) (2022) 23981–23994.
[27] H. Yu, T. Taleb, J. Zhang, Deep reinforcement learning based deterministic routing and scheduling for mixed-criticality flows, IEEE Trans. Ind. Inform. 19 (8) (2023) 8806–8816.
[29] IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams, IEEE Std 802.1Qav-2009, IEEE, Piscataway, NJ, 2010, http://dx.doi.org/10.1109/IEEESTD.2009.5375704, (Amendment to IEEE Std 802.1Q-2005).
[30] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34: Asynchronous Traffic Shaping, IEEE Std 802.1Qcr-2020, IEEE, Piscataway, NJ, 2020, http://dx.doi.org/10.1109/IEEESTD.2020.9253013, (Amendment to IEEE Std 802.1Q-2018).
[31] IEEE Standard for Local and Metropolitan Area Networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP), IEEE Std 802.1Qat-2010, 1–119, IEEE, Piscataway, NJ, 2010, http://dx.doi.org/10.1109/IEEESTD.2010.5594972, (Revision of IEEE Std 802.1Q-2005).

[32] C.J.C.H. Watkins, Learning from Delayed Rewards, King's College, Cambridge United Kingdom, 1989.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, 2013, arXiv preprint arXiv:1312.5602.

[34] S. Guadarrama, A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, E. Brevdo, TF-agents: A reliable, scalable and easy to use TensorFlow library for contextual bandits and reinforcement learning, 2018, URL https://github.com/tensorflow/agents. (Accessed: May 2023).

[35] A. Berisa, L. Zhao, S.S. Craciunas, M. Ashjaei, S. Mubeen, M. Daneshtalab, M. Sjödin, AVB-aware routing and scheduling for critical traffic in time-sensitive networks with preemption, in: Proceedings of the 30th International Conference on Real-Time Networks and Systems, RTNS 2022, 2022.

[36] L. Zhao, P. Pop, S.S. Craciunas, Worst-case latency analysis for IEEE 802.1Qbv time sensitive networks using network calculus, IEEE Access 6 (2018) 41803–41815.

[37] L. Zhao, P. Pop, Q. Li, J. Chen, H. Xiong, Timing analysis of rate-constrained traffic in ttethernet using network calculus, Real-Time Syst. 53 (2) (2017) 254–287.

[38] IEEE Standard for Local and Metropolitan Area Networks –Frame Replication and Elimination for Reliability, IEEE Std 802.1CB, 1–102, IEEE, Piscataway, NJ, 2017, http://dx.doi.org/10.1109/IEEESTD.2017.8091139, IEEE Std 802.1CB-2017.

[39] A. Ademaj, D. Puffer, D. Bruckner, G. Ditzel, L. Leurs, M.-P. Stanica, P. Didier, R. Hummen, R. Blair, T. Enzinger, Time sensitive networks for flexible manufacturing testbed characterization and mapping of converged traffic types, 2019, URL https://hub.iiconsortium.org/portal/Whitepapers/5eb04d87d2df3f001102b6fe. (Accessed: May 2023).

[40] D. Pannell, AVB - generation 2 latency improvement options - IEEE 802, 2011, URL https://www.ieee802.org/1/files/public/docs2011/new-avb-pannell-latency-options-1111-v2.pdf. (Accessed: May 2023).

Junhong Min received his B.S. degree from the School of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea, in 2021. He is currently continuing his study towards M.S. degree in the Department of Computer Science and Engineering at Chung-Ang University. He is also a research assistant at the Networked Systems Laboratory (NSL) led by Dr. Jeongyeup Paek, with research interests in time sensitive networking.

Yongjun Kim received his B.S. and M.S. degree from the Department of Computer Science and Engineering at Chung-Ang University, Seoul, Republic of Korea in 2019 and 2021, respectively. He was also a research assistant at the Networked Systems Laboratory (NSL) led by Dr. Jeongyeup Paek, with research interests in time sensitive networking. He is currently a software engineer at TmaxSoft.

Moonbeom Kim received his B.S. degree in Computer and Information Communications Engineering from Hongik University in 2017, and the M.S. degree in Computer Science and Engineering from Chung-Ang University, Seoul, Republic of Korea, in 2020. He is currently pursuing the Ph.D. degree in Computer Science and Engineering. He is also a research assistant with the Networked Systems Laboratory (NSL) led by Dr. Jeongyeup Paek, with research interests in wireless networking, localization, and time-sensitive networking.

Jeongyeup Paek received his B.S. degree from Seoul National University in 2003 and his M.S. degree from University of Southern California in 2005, both in Electrical Engineering. He then received his Ph.D. degree in Computer Science from the University of Southern California (USC) in 2010. He worked at Deutsche Telekom Inc. R&D Labs USA as a research intern in 2010, and then joined Cisco Systems Inc. in 2011 where he was a Technical Leader in the Internet of Things Group (IoTG), Connected Energy Networks Business Unit (CENBU, formerly the Smart Grid BU). In 2014, he was with the Hongik University, Department of Computer Information Communication as an assistant professor. Jeongyeup Paek is currently an associate professor at Chung-Ang University, School of Computer Science and Engineering, Seoul, Republic of Korea since 2015. He is on the editorial board of Journal of Communications and Networks (JCN) and Sensors. He is an IEEE senior member and an ACM member.

Ramesh Govindan is the Northrop Grumman Chair in Engineering and Professor of Computer Science and Electrical Engineering at the University of Southern California. Dr. Govindan received the B.Tech degree from the Indian Institute of Technology at Madras, and the M.S. and Ph.D. degrees from the University of California at Berkeley. Prior to joining USC he was a member of the technical staff at Bell Communications Research, and a project leader at USC's Information Sciences Institute and at the International Computer Science Institute at Berkeley. Dr. Govindan's research has focused on scalable and robust routing infrastructures in large networks such as the Internet, on the structural properties of the Internet, and on the architectures and programming systems for wireless and mobile networks. He is a Fellow of the ACM and of the IEEE, the recipient of the 2018 IEEE Internet Award, a former Editor-in-Chief of the IEEE Transactions on Mobile Computing, and a Distinguished Alumnus of the Indian Institute of Technology, Madras.