Train Once Apply Anywhere: Effective Scheduling for Network Function Chains Running on FUMES

Marcel Blöcher
SAP *
m.bloecher@me.com

Nils Nedderhut
vivenu *
nils@nedderhut.de

Pavel Chuprikov
USI
chuprp@usi.ch

Ramin Khalili
Huawei Technologies
ramin.khalili@huawei.com

Patrick Eugster
USI
eugstp@usi.ch

Lin Wang
Paderborn University
lin.wang@upb.de

***Abstract*—The emergence of network function virtualization has enabled network function chaining as a flexible approach for building complex network services. However, the high degree of flexibility envisioned for orchestrating network function chains introduces several challenges to support dynamism in workloads and the environment necessary for their realization. Existing works mostly consider supporting dynamism by re-adjusting provisioning of network function instances, incurring reaction times that are prohibitively high in practice. Existing solutions to dynamic packet scheduling rely on centralized schedulers and a priori knowledge of traffic characteristics, and cannot handle changes in the environment like link failures.**

We fill this gap by presenting FUMES, a reinforcement learning based distributed agent design for the runtime scheduling problem of assigning packets undergoing treatment by network function chains to network function instances. Our design consists of multiple distributed agents that cooperatively work on the scheduling problem. A key design choice enables agents, once trained, to be applicable for unknown chains and traffic patterns including branching, and different environments including link failures. The paper presents the system design and shows its suitability for realistic deployments. We empirically compare FUMES with state-of-the-art runtime scheduling solutions showing improved scheduling decisions at lower server capacity.

I. Introduction

The emergence of NFV (network function virtualization) has enabled the vision of a converged network architecture, spanning data centers, carrier networks, and edge, accessing dynamically created services using resources distributed across the network. Scheduling such network services end-to-end, with packets undergoing treatment by combinations of NFs (network functions) orchestrated into NFCs (network function chains) [1], remains an open challenging problem [2]–[4].

A. Real-life Constraints

The main challenge in conceiving NFCs in a way realizing the vision of flexible services for real-life deployments is to support the necessary *dynamism*, especially when considering support for wireless communication in 6G. More precisely, solutions must address the following requirements:

Dynamic workload **[DYN-WORK]:** Packet arrivals for NFCs do not necessarily follow some fixed, known, rate or distribution, but can exhibit more fundamental unforeseen workload variations. This includes branching decisions, as required for expressive NF orchestrations.

Dynamic environment **[DYN-ENV]:** Similarly, execution environments are not static. Networks may get extended with new links or improved link latencies, and servers can get added. Inversely, resource reductions including outages, *e.g.*, link failures, need to be accommodated. Testing and adjusting is often not even possible in real-life settings ahead of production time. Adaptations must happen automatically on the fly with minimal disruption.

B. Lack of Dynamic Solutions

We know of no solution capable of addressing these needs among existing works. Several works periodically adjust the *deployment* of NFIs (network function instances) or individual VNFs (virtualized network functions) and their resource assignments to changes in network traffic and topology, as required for future carrier networks [5]–[10]. However, these solutions are too coarse-grained for [DYN-WORK], *e.g.*, with adaptations taking seconds to take effect [10], incurring complexities prohibitive for real-time application [8], and involving disruptive NFI migration [6]. Solutions for scheduling packets for VNFs or NFCs mostly perform network-wide *centralized* scheduling, assuming a scheduler with global knowledge of the network and *advance* knowledge of statistical information – often even assuming static bandwidth requirements for network flows [9], [11]–[20]. The few solutions considering dynamic flow demands [5], [6], [9], make assumptions preventing real-life dynamic deployments as captured by [DYN-WORK], like knowledge of input and output traffic volume of all NFIs [9], of flows' nominal and maximum volumes [6], or of flow rates [21]. The closest matching work is STEAM [22], which treats traffic scheduling entirely as a runtime problem without a priori knowledge of traffic characteristics. However, STEAM supports neither expressive NFC orchestrations with branching necessary for [DYN-WORK], nor handles environmental changes as per [DYN-ENV] like link failures.

C. FUMES

This paper presents FUMES (network function chaining with reinforcement learning incorporating state), the first NFC packet scheduling solution fully addressing [DYN-WORK] and [DYN-ENV]. We use RL (reinforcement learning) to solve the MDP (Markov decision process) of deciding *on the fly* for a given packet which NFI for the next NF in the packet's NFC to process the packet at. More precisely we propose a *distributed*

* Work done at TU Darmstadt.

scheduler with a MARL (multi-agent reinforcement learning) design, with an own independent agent for each NFCC (network function chain coordinator) managing ingress and egress at a site, thus becoming an instance of Markov game [23].

Every agent uses packet metadata and collected network state to calculate a state table and generate an action vector represented by an array of probabilities for all possible actions, following a policy. An agent draws a random decision based on this probability distribution to decide which NFI to send a given packet to next for processing. Each time an agent is involved in scheduling decisions, the agent calculates a reward to improve its policy; the reward function uses several parameters including the current processing stage in the NFC and processing latency, and the estimated latency to reach the next NFCC. By adopting techniques such as padding and state shuffling, and by leveraging gossip-based communication for state sharing, FUMES is able to effectively adapt to changes in both workload ([DYN-WORK]) and environment ([DYN-ENV]), while minimizing synchronization across agents and sites.

D. Contributions and Roadmap

After specifying the model of multi-site NFC execution considered, and the dynamic packet-level scheduling problem for that model (§ II), this paper makes the following contributions:

1) for ease of presentation only, we first conceptually map the considered scheduling problem to an MDP and show how to solve it via RL using simplifying assumptions, such as a single oracle agent with full visibility over the state of all NFCCs taking decisions (§ III);
2) we lift the assumptions and map the problem to a realistic, distributed setup with multiple sites, and individual agents for all NFCCs, leveraging gossiping for state sharing and a carefully designed distributed reward function (§ IV);
3) we empirically evaluate FUMES, comparing it against an extension of STEAM [22] to support branching as well as link failures, and a greedy heuristic similarly adapted from prior works. We show that FUMES achieves higher success rates and lower latency, and uses less server capacity, especially under workload spikes, network link failures, and unpredictable branching, demonstrating its ability to achieve [DYN-WORK] and [DYN-ENV] (§ V).

§ VI presents related work, and § VII draws conclusions.

II. Model and Problem

This section provides a comprehensive system model for runtime NFC packet scheduling in a distributed system.

A. System Model

1) Infrastructure: We consider a distributed system with several geographically distributed computing sites, *e.g.*, edge or cloud data centers, holding servers for running NFIs. We denote sites by capital letters (*e.g.*, A, B, C). Each site has an NFC coordinator (NFCC), responsible for forwarding traffic within its site and to other sites; each NFCC runs a classifier and a scheduler. Examples of NFCCs include service function forwarders in SFC architectures [24] or front-end servers in data centers [25]. We denote the coordinator of a site A by A^{C}. The NFCC is seconded by a set of servers running NFIs, as depicted in Fig. 1, denoted by A_1, A_2, *etc.*

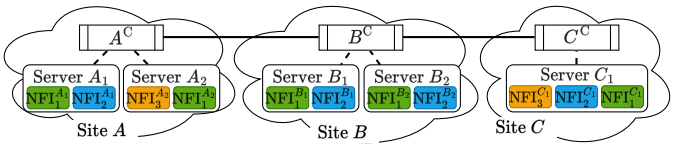


Fig. 1: NFC scenario with 3 sites, inter-connected through their NFCCs, and 3 different types of NFs with multiple instances.

We assume that network planning, as discussed in [26], [27] for different use cases, is performed beforehand, with enough capacity assigned to links interconnecting NFCCs. Generally, these links are not a bottleneck; however, the communication over these links can be unreliable (fail at runtime for unknown time periods) and is subject to latency. The propagation latency of a link is a random variable with finite mean (based on the physical length of the link) and variance.

2) NFIs: An NF is a piece (type) of processing logic applied to network packets, while an NFI is a concrete instantiation of such an NF on a server. Multiple instances of the same NF might be deployed at a same site. We denote by NF_i the NF of type $i = 1$, 2, etc., and by $\mathrm{NFI}_i^{A_k}$ an instance of NF type i deployed on server A_k at site A.

We consider that NFIs are already deployed in the network. That is, we consider a capacity planning scheduler that runs periodically (minutes granularity) and updates the NFI installation, by using any solution proposed in the literature (*e.g.*, [12], [14]). In contrast to these solutions, however, NFIs are not pre-assigned resources or traffic. Our scheduler decides entirely at runtime where to send a packet and how many resources to assign to its processing. Naturally each NFI has a local buffer to store incoming packets. We also assume that the processing capacity of a server is shared among all hosted NFIs according to a given policy. The above-mentioned assumptions are not restrictive, representing many real-life use cases [28].

3) Network traffic: The network traffic is composed of many flows originating from different users connected to the network. The ingress and egress nodes of a flow are determined from the flow's source and destination addresses. A flow's packets should go through an ordered chain of NFs, determined by packet classification. This classification can be performed at NFCCs, by the classifier deployed at these nodes. The classification information can be embedded in a packet's header, which also maintains the packet's processing stage, specifying by which NF in its NFC it is to be processed next.

We apply a generic definition of NFC as a directed graph with branches, each edge specifying an NF type [24]. The example NFC in Fig. 2 has two or three NFs, depending on the branching decision after NF_1. Any branching decision is made at arrival of the first packet of a flow at the corresponding hop of the chain; all remaining

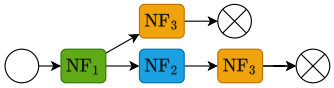


Fig. 2: NFC with 2 or 3 steps depending on branching.

packets of a flow take the same branch. In addition, each NFC is given a set of QoS metrics that the handling of packets undergoing that NFC has to conform to, which in our considered scenarios contains a maximum end-to-end delay. An NF in the chain can be handled by any of its instances.

B. Service Scheduling Problem

We consider instances of the service scheduler running at each NFCC. The service scheduling problem consists in deciding at runtime where to route a packet next, *i.e.*, how to serve the next hop in the chain of a packet, knowing the chain undergone by the packet and its current position in that chain.[1] The decision is made by the scheduler instance receiving the packet. We assume no buffering at the scheduler, so it is to operate at line rate. For each packet, the end-to-end delay is the sum of the queuing and service delays at NFIs along its NFC path and propagation delays. Our objective is to maximize the system's processing goodput (the rate of packets successfully processed over respective chains within their delay budgets), while constraining the average delay experienced by packets.

C. Challenges

The runtime traffic scheduling problem is hard to solve due to the distributed setup and strict requirements posed by the high dynamics ([DYN-WORK] and [DYN-ENV], *cf*. § I-A). *Centralized* schedulers adopting corresponding optimal solvers are impractical due to the rigid latency requirement on scheduling decisions. Distributed schedulers applying *heuristics* for decision-making are typically optimized for specific static scenarios, require parameter tuning, and fall short of generalizing to dynamic scenarios with uncertainty (*e.g.*, unpredictable workload variations, network link failures). Hence we explore learning-based approaches which have shown great potential in dealing with dynamics and uncertainty for decision-making problems. However, non-trivial domain-specific designs are required for such a solution to succeed.

III. Problem Transformation

This section introduces the building blocks for our FUMES scheduler detailed shortly in § IV. They map the aforementioned NFC runtime traffic scheduling problem to an MDP and use RL to solve the MDP. For ease of exposition only, we first simplify the scheduling problem by assuming a single oracle controller with full visibility over the state of all NFCCs and their NFIs. In § IV we relax this assumption and discuss how to use MARL to address the challenges posed by distribution.

A. Problem Mapping Overview

Fig. 3 gives an overview of the problem mapping focusing on the lower branch of Fig. 2 for simplicity. The oracle controller collects real-time state information from the network

[1]Here, we consider per-packet scheduling, where packets from the *same* flow can be scheduled separately over *different* NFIs with state synchronized by an existing mechanism (*e.g.*, [29]). Our solution can be adapted to more constrained settings where consecutive packets of a flow or flowlet need to be served by the same NFI (*e.g.*, due to high state synchronization latency across NFIs) [30]. We leave such an extension to future work.

TABLE I: Notation used.

Symbol	Description	
α	Overload normalization	Used in § III and § IV
β	Reward scaling factor	
$\mathrm{SP}_i^{a_k}[b]$	State column *pressure* view from site b on NF_i	
$\mathrm{SB}_i^{a_k}[b]$	State column *background* view from site b on NF_i	
$\mathrm{SS}^{a_k}[b]$	State column *sever speed* view from site b on NF_i	
$\mathrm{SL}^{a_k}[b]$	State column *latency* view from site b on NF_i	
$\mathrm{SH}_i^{a_k}[b]$	State column *probability next* view from site b on NF_i	
$\mathrm{AA}_i^{a_k}$	Agent's action vector's probability of taking $\mathrm{NFI}_i^{a_k}$	
$\mathsf{d}(x,y)$	Estimated latency from x to y (server/NFCCs)	
PR	Reward of packet	w.r.t. current packet
PD	Deadline of the chain	
PL	Current latency of the packet	
PT	Remaining time for packet	
PQ	QoS (%) when packet arrives at egress	
FC^{a_k}	Processing capacity of server a_k	
FL^{a_k}	Network latency to reach a_k	
$\mathrm{FW}_i^{a_k}$	Queueing time at $\mathrm{NFI}_i^{a_k}$ w/o considering other NFIs	
$\widehat{\mathrm{FW}}_i^{a_k}$	Queueing time at $\mathrm{NFI}_i^{a_k}$ w/ considering other NFIs	
$\mathrm{FP}_i^{a_k}$	Time (ms) to process one packet when NFI at full speed	
$\mathrm{FN}_i^{a_k}$	# of packets queued at or on the way to $\mathrm{NFI}_i^{a_k}$ NFI	
PM	# of NFs in the chain, in case of branches, the longest path	Used in § IV and § V
PH	Weighted processing progress of the chain $\in [0,1]$	
$\widehat{\mathrm{PQ}}$	Expected QoS $\in [0,1]$ when packet arrives at egress	
AT	Number of max scheduling targets FUMES is trained for	

and implements agents making packet scheduling decisions. When a new packet arrives at an NFCC, the NFCC classifies the packet to assign it an NFC given the packet's metadata. Based on the metadata and the collected network state, the controller selects the agent for the next NF and calculates a state table (§ III-C). With that table as input, the agent at the controller generates an action vector represented by an array of probabilities for all possible actions, following a pre-trained policy. The array has an entry for each NFI in the network indicating the probability of forwarding the packet to that NFI. The agent draws a random NFI following this probability distribution and returns the decision to the NFCC, which forwards the packet to that NFI. Once an NFI finished processing a packet, it sends the packet back to the NFI's NFCC, repeating the above procedure for further processing.

When a packet has been processed by the last NF in its NFC, the controller collects the packet's end-to-end latency (defined in § II-B). The controller then calculates a reward (detailed in § III-B) comparing this latency to the deadline specified for the NFC. This reward is then fed back to all agents that took scheduling decisions for the specific packet, to continuously train their policy for generating the action vector.

Table I summarizes notation and terminology used in the following. $\mathsf{min}(R)$ and $\mathsf{max}(R)$ return the smallest and largest element of set R respectively. For brevity we omit the set notation $\{...\}$ for R but may add variables quantified over for constructing R, *e.g.*, $\mathsf{min}_z(r(z))$ for $R = \{r(z)\}_z$. We use i, j as (abstract) identifiers for NF types, and k, l for servers. a, b, c, e are used for sites with e reserved for the egress site. x, y represent both coordinators (*e.g.*, b^{C}) and servers (*e.g.*, b_l).

B. Actions

Each time a packet arrives at an NFCC, the agent makes a scheduling decision for the next stage. Possible actions are

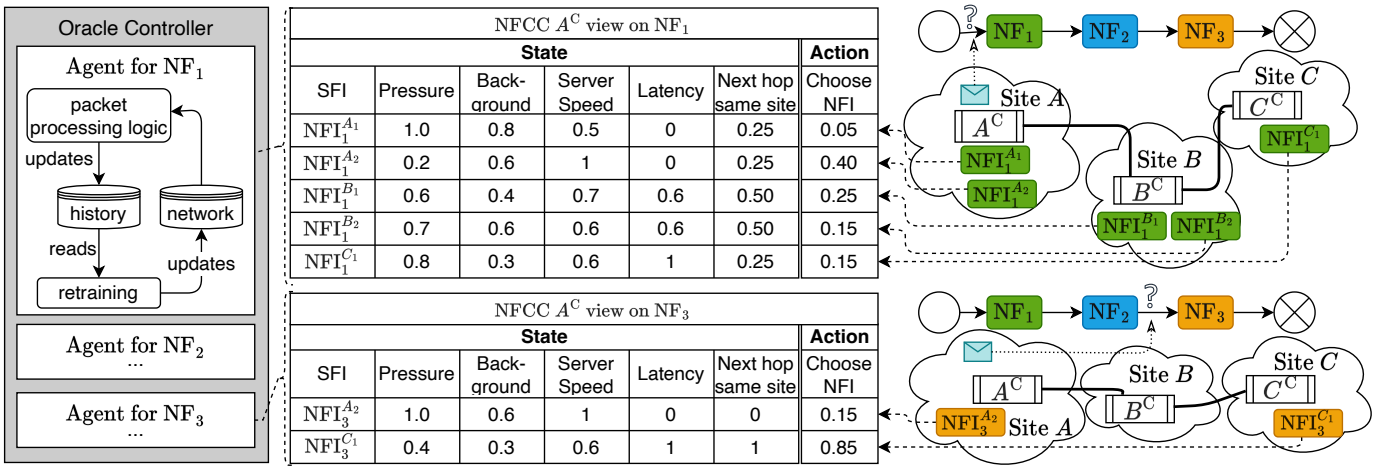


Fig. 3: Perspective of A^C: This example shows the state table with 5 rows for a new packet arrived at A^C with next hop of type NF_1, and the state table with 2 rows for a packet at A^C that awaits processing by NF_3. (w/o servers for sake of readability.)

(a) drop the packet upon QoS timeout,
(b) send the packet to the desired egress site,
(c) serve the packet at an NFI locally (the same site), or
(d) serve the packet at an NFI remotely (at a different site).

Cases (a) and (b) do not need scheduling logic; the agent simply drops the packet, or sends it to the desired egress site along the shortest path respectively. In case (c), the NFCC forwards the packet directly to the selected NFI, which processes it and after its done sends it back to the NFCC for further decisions. In case (d), the packet is redirected to the remote site and then immediately forwarded (without scheduling by the remote agent) to the selected NFI. After processing at the NFI, the packet goes to the respective remote NFCC which takes over scheduling for the chain's next stage.

C. State

The agent considers five state metrics to take as input for its scheduling decisions: For the view of site b's agent on NF_i and server a_k, we define the *pressure* ($SP_i^{a_k}[b]$), *background* ($SB_i^{a_k}[b]$), *server speed* ($SS^{a_k}[b]$), *latency* ($SL^{a_k}[b]$), and *probability next hop* ($SH_i^{a_k}[b]$). These are detailed shortly.

We define network latency FL^{b_k} – the estimated network latency to send the packet from NFCC a^C where the packet is located at, to NFCC b^C local to the next desired $NFI_i^{b_k}$ ($a = b$ is possible), plus the estimated network latency to send the packet from b^C to b_k – as:

$$FL^{b_k} = \mathsf{d}(a^C, b^C) + \mathsf{d}(b^C, b_k), \quad \text{packet at site } a. \tag{1}$$

Furthermore, we define the remaining time of the current packet $PT = PD - PL$, *i.e.*, it is the difference between the deadline PD of the chain and the current latency PL of the packet. The achieved QoS of a packet is the ratio $PQ = PL/PD$ calculated when the packet arrives at the desired egress.

All state metrics are normalized to values in $[0, 1]$. However, some situations might represent overload scenarios. These may only be visible as part of the performance approximation, or real overload scenarios where NFIs cannot cope with the arriving packets to satisfy all chain deadlines. This is why we introduce the scaling factor α, which is set to $0 < \alpha < 1$, such that the range of values considered as *normal* is encoded as $[0, \alpha]$ (instead of $[0, 1]$), and *overloaded* situations are in $(\alpha, 1]$. The following formulas use the scaling factor α and for values > 1 adopt 1. This normalization is applied in all state formulas except for the *server speed* and *probability next* column, since these two metrics are unaffected by overload.

1) Pressure: The current pressure $SP_i^{a_k}[b]$ of $NFI_i^{a_k}$ is computed from the NFI's processing rate and the number of pending packets. To calculate the pressure, the agent uses the expected processing time of a packet of the respective NF, the expected processing capacity of the NFI, and the number of packets that the NFI has not processed yet. Eq. 2 defines the pressure metric ($SP_i^{a_k}[b]$), considering the remaining time for the packet in relation to the expected waiting time $FW_i^{a_k}$:

$$SP_i^{a_k}[b] = \min\left(1, \alpha \times {}^{FW_i^{a_k}}/\left(\min\left(PT, \max_{c,l}\left(FW_i^{c_l}\right)\right)\right)\right). \tag{2}$$

The expected waiting time $FW_i^{a_k}$ in turn is given as

$$FW_i^{a_k} = \max\left(0, FP_i^{a_k} \times FN_i^{a_k} - FL^{a_k}\right). \tag{3}$$

2) Background: A server a_k's background $SB_i^{a_k}[b]$ is similar to its pressure ($SP_i^{a_k}[b]$), but considers all the server's NFIs:

$$SB_i^{a_k}[b] = \min\left(1, \alpha \times {}^{\widehat{FW}_i^{a_k}}/\left(\min\left(PT, \max_{c,l}\left(\widehat{FW}_i^{c_l}\right)\right)\right)\right). \tag{4}$$

Eq. 5 in turn calculates the expected waiting time of the packet, considering all packets that are processed or are on the way to any NFI of that particular server a_k as

$$\widehat{FW}_i^{a_k} = \max\left(0, \sum\nolimits_m \left(FP_m^{a_k} \times FN_m^{a_k}\right) - FL^{a_k}\right). \tag{5}$$

3) Server speed: The relative speed of a server a_k, *i.e.*, its processing capacity, normalized with respect to the maximal processing capacity among all the servers is given as:

$$SS^{a_k}[b] = {}^{FC^{a_k}}/\max_{c,l}\left(FC^{c_l}\right). \tag{6}$$

4) Latency: The expected relative latency for a packet to travel from the current NFCC (a^C) to the desired $NFI_i^{a_k}$, including the estimated latency towards the egress also for intermediate stages of the chain, is defined as

$$SL^{a_k}[b] = \min\left(1, \alpha \times \left(FL^{a_k} + \mathsf{d}(a_k, a^C) + \mathsf{d}(a^C, e^C)\right)/PT\right) \tag{7}$$

```
for nfcc in self._all_nfcc: # iterate NFC coordinators
 for nf in self._all_nf: # iterate NFs (not instances)
  state = self._state(origin=nfcc, nf=nf, summarize=True)
  self._next_hop_prob[nfcc][nf] = self._action(state)
```

Listing 1: Calculation of cached $\sum_k \text{AA}_i^{a_k}$ by summarizing NFIs of same NF for each site.

for a packet at site a with egress at site e.

5) Probability next hop: $\text{SH}_i^{a_k}[b]$ denotes the probability, for a packet at $\text{NFI}_i^{a_k}$, that the next stage of the chain will be at the current site a. *I.e.*, either the next hop NFI NF_j is at a, or the desired egress e^{C} is such that $e = a$. This back-propagation mechanism achieves scheduling decisions considering more than a single NFC stage. In the example of Fig. 1 and chain Fig. 2, the next hop action vector for the second stage in the lower branch (NF_2) may be $(0.6, 0, 0.4)$, denoting 60%, 0%, and 40%, probability of serving NF_3 for this packet at site A, B, and C, respectively.

Considering a single branch of the current chain of the packet, $\widehat{\text{SH}}_i^{a_k}[b]$ gives the next hop probability as

$$\widehat{\text{SH}}_i^{a_k}[b] = \begin{cases} 1 & \text{if next hop egress at site } e = a, \\ 0 & \text{if next hop egress at site } e \neq a, \\ \sum_l \text{AA}_j^{a_l} / \sum_{c,l} \text{AA}_j^{c_l} & \text{if next hop is of type } \text{NF}_j. \end{cases} \quad (8)$$

$\text{AA}_i^{a_k}$ is the agent's action vector's probability to take $\text{NFI}_i^{a_k}$. There are two cases to differentiate: The next stage of the chain is (a) the desired egress, or (b) a normal NF. If the next stage is the desired egress, the agent can easily fill up the column with probabilities (0 or 1) based on whether the egress is the same site or not (first two cases in Eq. 8).

For the state table, $\text{SH}_i^{a_k}[b]$ combines all individual branches (if there are multiple) using the average of all $\widehat{\text{SH}}_i^{a_k}[b]$. Otherwise, if there are no branches, $\text{SH}_i^{a_k}[b] = \widehat{\text{SH}}_i^{a_k}[b]$.

The look ahead with $\text{AA}_i^{a_k}$ requires additional inference by the agent. However, simple caching can be used for the agent to quickly (without additional inference delays) inject the values for the *probability next* column while constructing the state for a scheduling attempt. In total, the agent needs $|\text{NFs}| \times |\text{sites}|$ cached action vectors. Listing 1 shows FUMES pseudo code for calculating the cached next hop action vectors.

D. Reward

The main goal for the runtime NFC scheduling problem is to achieve high success rate of packet delivery (*i.e.*, maximize the ratio of packets processed by the system before reaching the respective NFC deadlines). Beyond that, we aim to minimize the amount of time a packet has to stay in the system as the secondary goal. The design of a reward function should positively reward actions taken by the agent that affect the goals in the long run. The most straightforward way is to incorporate the goals directly into a reward function that is calculated for each action taken. However, in our setting, the success rate and delay of a packet are known only when it has been processed *completely* by its NFC. Hence, we choose a *sparse* (or *delayed*) reward environment [31], assigning a reward to the agent after multiple actions. Our reward (PR) function is defined as follows:

$$\text{PR} = \begin{cases} 1 - {}^{(\text{PL} \times \beta)}/_{\text{PD}} & \text{if successfully delivered,} \\ -1 & \text{otherwise (timeout).} \end{cases} \quad (9)$$

β is a scaling factor to adjust the impact of the achieved PQ ratio. For the centralized agent assumed in this section only, this reward can be calculated globally when a packet is done being processed and leaves the network. The reward is distributed to all the actions that have been taken for the packet, with the help of a database maintaining the decisions the agent has taken for each packet. Clearly, the database can quickly become a performance bottleneck. The next section shows how to address such distribution issues.

IV. FUMES: Distributed Cooperative Scheduling

This section presents FUMES, a distributed scheduler with a MARL design combining the building blocks from § III. We first summarize challenges of moving from those blocks to a distributed setting, then present how FUMES addresses them.

A. Overview: Generalization Challenges

The high dynamicity and uncertainty of the network challenge the approach of § III in real-life setups w.r.t.:

1) Scalability: The network consists of multiple sites each with its own NFCC and set of NFIs. The centralized agent assumed in § III can not scale to such a large setup. We present a scalable MARL approach employing a dedicated agent for each site (§ IV-B) and using a gossip protocol (§ IV-D) to diffuse state information across sites.

2) Varying number of servers: Our state definition includes per-NFI and per-host information. The size of the state table thus depends on the number of servers and NFIs in the system, whilst in real-life scenarios servers and NFIs can get added/removed following workload changes. A trained agent with a fixed number of servers and NFIs can not be applied then. We present techniques including padding (§ IV-C1) and shuffling (§ IV-C2) of state table rows addressing this issue.

3) Link dynamics: In real life, link failures are alas common. Also, new links can get added to the network or link capacities adjusted following capacity planning. The agent must detect these changes in a timely manner and incorporate them in decision making. We present a gossip protocol performing latency approximation to detect such link dynamics (§ IV-D).

4) Multiple agents: The training approach described in § III uses one agent per site training independently of others. However, when an agent is in training mode, other sites perform scheduling decisions which impact the performance of the local site. Moreover the reward calculation of § III requires a global view across all sites, and packets leave the network not only at a single site, but at all sites. § IV-E presents a learning approach incorporating intermediate processing steps and achieving distributed reward by QoS estimation.

B. MARL Design

The core design choice of FUMES is its distributed MARL (multi-agent reinforcement learning) approach. Each site is

managed solely by the corresponding NFCC with one site-local independent RL agent that has its own actor-critic network. The scheduling problem in FUMES is similar to the one described in § III, but differs crucially with respect to the state visibility each agent has, its action responsibility, and how the training is set up (distributed and asynchronous).

Each FUMES agent performs (a) scheduling decisions to serve packets at site-local NFIs, and (b) *redirection* decisions for NFIs at remote sites. However, differently from remote decisions in § III, redirected packets are sent to remote NFCCs and then picked up again for scheduling there. As depicted in Fig. 4, A^{C} sees all its site-local NFIs, but for the other sites only a single summarized NFI for each NF.

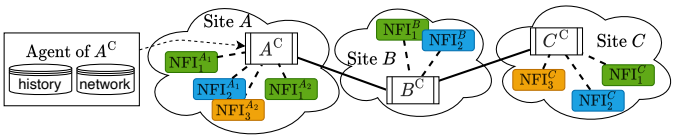


Fig. 4: Perspective of FUMES's agent at A^{C}, *cf*. Fig. 1

C. State

Similar to the definition of the state table in § III, a FUMES agent has a row for each site-local NFI. However, for each remote site that has NFIs of some NFs, only one summarizing row is retained. This significantly reduces the size of the state table which positively affects training time. Yet the problem of different required state table sizes (for different NFs) remains. Instead of training (and using) multiple agents with different state sizes to handle different numbers of NFI instances, FUMES uses two techniques to address the problem together.

1) Padding: The first technique chooses a single state table size for all scheduling problems (*i.e.*, all NFs) of an agent. For this, FUMES chooses the largest number of site-local NFIs for any NF, and adds the number of other sites that have NF-matching NFIs. For the running example of Fig. 1, A^{C} would need 4 rows for NFI_1, 3 rows for NFI_2, and 2 rows for NFI_3, as shown in Fig. 4. Thus FUMES chooses a size of 4. When constructing the state for a NF which needs fewer rows than configured, FUMES fills remaining rows with zeros.

2) Shuffling: The second technique shuffles all rows of the state-action table, such that the corresponding rows of the states and actions stay aligned. Shuffling has the goal to prevent the agent from training topology-related information as part of the row number (which would be wrong since rows are used across NFs), and to equally weigh all rows representing a nonexistent entry (zero-ed rows), a site-local NFI, or an entry representing per-site summarized NFIs.

The benefits of doing so and of the way how FUMES constructs the state and action space are manifold, as we show later in the evaluation (§ V). The agent is trained in a generic way which decouples it from the configuration of NFCs, installed NFIs and set of NFs, network topology and number of NFCCs, and available servers and their capacity. The only restriction is the maximum number of supported scheduling targets AT that can be considered for scheduling at a single scheduling attempt. In case one wants to use a trained FUMES instance for a setup with a larger number of NFIs than what it was trained for, a possible approach is to use a subset of all NFIs based on sorted *background* state column.

D. Gossip Protocol

FUMES needs summarized NFI information of remote sites when constructing state tables. It thus uses a gossip protocol to share summarized NFI updates with neighboring NFCCs.

1) State diffusion: Each FUMES agent at a regular time interval sends a gossip update to its directly neighboring NFCCs with the following information. For each NF, all site local NFIs are summarized to represent a conceptual single NFI running on a single large server of the summed capacity. This information is sufficient to construct the state table analogously to how it is defined in § III-C, but with a single row entry per remote site. Each gossip message holds a site-local gossip interval counter, incremented for each message.

When a FUMES agent receives a gossip message for a remote site, it checks if the gossip interval counter is higher than the largest previously seen value for that site. If so, the agent forwards the message to all direct neighbors (except the sending one), and updates its local database with the NFI summaries and the new interval counter for that site.

The gossip protocol could also be used to support dynamic processing speeds of NFIs and dynamic server capacity or set of available servers, without modifying FUMES, simply because FUMES' gossip protocol hides these details and shares only the summarized statistics.

2) Latency approximation: FUMES furthermore uses gossiping to estimate inter-site latencies (shortest path between sites), which it uses in the state table calculation (§ III-C). That is, gossip messages piggyback timing information to estimate end-to-end link latency, allowing to better approximate up-to-date link latencies, and cope with variable link latencies and sporadic link downtimes. We refer to common methods (*e.g.*, NTP in RFC 5905 [32]) for link latency approximation across nodes without synchronized clocks – our implementation used in § V uses clocks synchronized across NFCCs.

FUMES uses a gossip interval of $2ms$. The interval is a hyperparameter which we consider to be stable across various setups – as long as it is aligned with the inter-site latencies. We consider $2ms$ to be sufficient for real-life inter-site latencies without adding too much overhead on top of normal traffic.

E. Distributed Learning and Reward

To speed up training, FUMES learns not only from packets that leave the network because of egress or timeout events, but from all packets that leave an NFCC's site. To that end FUMES creates a reward tuple each time a packet leaves the network of an NFCC, which happens (a) when a packet is sent to another remote NFCC for processing, (b) in case of a timeout drop event, or (c) when a packet leaves the network because the NFCC was the packet's destination egress.

For calculating reward FUMES uses several pieces of information, namely the packet's (1) current delay, (2) NFC

and respective deadline, (3) current processing stage in its NFC, (4) target egress NFCC, as well as (5) the estimated end-to-end network latency to reach the egress starting from the current NFCC. FUMES uses this information to estimate the expected QoS when taking the current delay into account w.r.t. processing progress of the corresponding NFC. A simple linear approximation of the processing progress (*e.g.*, 2 out of 4 completed NFs mean 50% progress) would not account for different processing rates of different NFIs. Therefore, FUMES uses PH, the *weighted* approximation of processing progress that considers the total available processing rate of each NFI. FUMES combines all these terms to calculate the *expected* QoS (processing latency) of a packet it would get at the egress b^{C} when leaving the site of the local agent at a^{C}:

$$\widehat{\mathrm{PQ}} = \begin{cases} (\mathrm{PL}/\mathrm{PH} + \mathsf{d}(a^{\mathrm{C}}, b^{\mathrm{C}}))/\mathrm{PD} & \text{if PH} > 0, \\ (\mathrm{PL} \times \mathrm{PM} + \mathsf{d}(a^{\mathrm{C}}, b^{\mathrm{C}}))/\mathrm{PD} & \text{otherwise.} \end{cases} \tag{10}$$

$\widehat{\mathrm{PQ}}$ helps FUMES estimate how likely a packet is to be processed successfully in its NFC deadline. $\widehat{\mathrm{PQ}} \geq 1$ means the packet will likely be dropped due to timeout. Smaller values are better. FUMES uses $\widehat{\mathrm{PQ}}$ (Eq. 10) to calculate reward:

$$\mathrm{PR} = \begin{cases} 1 - \widehat{\mathrm{PQ}} \times \beta & \text{if } \widehat{\mathrm{PQ}} \leq 1, \\ -1 & \text{otherwise (expected timeout).} \end{cases} \tag{11}$$

We use an additional scaling factor β to reduce the sensitivity to QoS over successfully delivered packets.

V. Performance Evaluation

We conduct performance evaluation including real-life workload traces to address the following research questions:

Q1. What is the training convergence performance of the MARL design of FUMES (§ V-B)?

Q2. How well does FUMES perform in general, and how well does it dynamically adapt to changes in traffic models (*cf*. [DYN-WORK]; § V-C) or configurations of NFCCs and NFIs (*cf*. [DYN-ENV]; § V-D)?

Q3. What is the impact of entirely unseen NFC configurations and NFC branching (*cf*. [DYN-WORK]; § V-E)?

Q4. How robust is FUMES against (transient) link failures (*cf*. [DYN-ENV]; § V-F)?

A. Methodology

1) Workload: We use two topologies from a US and German network respectively, based on commonly used [33], [34] real-life traces from SNDlib [35] with randomly chosen amounts of servers per site around a given respective mean:

Abilene: 12 sites, 15 links, and a mean of 20 servers per site.
Nobel-Germany: 17 sites, 26 links, and a mean of 15 servers.

Every site acts as potential ingress and egress for all packets. We set inter-NFCC latencies following a Poisson distribution in relation to the geographical distance of the respective NFCCs. Link latencies within a site, *i.e.*, between the NFCC and the NFIs, follow a Poisson distribution with mean of $120\mu s$. Two workloads are used for the network traffic:

MDP: Bursty traffic with a two-state MDP traffic model simulating the arrival of new flows. The two states *high* and *low* have arrival rates $\lambda_h = 1/120\mu s$ and $\lambda_l = 1/24\mu s$, respectively. The transition probabilities are $p_{l\to h} = 0.56$ and $p_{h\to l} = 0.4$. Each flow follows a Poisson distribution with $\lambda = 1/800\mu s$ for packet arrival, with a total number of packets per flow following Poisson with $\lambda = 800$.

SND: Trace-driven workloads from SNDlib [35], namely Abilene and Nobel-Germany, for the respective topologies.

These traffic models do not provide NFC configurations, so we use the following two setups. Unless stated otherwise, we use NFCs without branching with 5 NFs with a processing rate per 1 unit of server capacity of $1s/80\mu s$ to $1s/200\mu s$. We use 5 NFCs: $(\mathrm{NF}_1, \mathrm{NF}_2)$, $(\mathrm{NF}_1, \mathrm{NF}_3, \mathrm{NF}_5)$, $(\mathrm{NF}_2, \mathrm{NF}_4)$, (NF_5), and $(\mathrm{NF}_3, \mathrm{NF}_4)$, with QoS deadlines $50ms$, $55ms$, $50ms$, $45ms$, and $50ms$ respectively. The experiment on unseen NFCs and branching uses another configuration detailed in § V-E. We randomly assign NFIs on servers such that each server has at least 2 NFIs, and $|\mathrm{NFIs}| < 0.6 \times |\mathrm{NFs}| \times |\mathrm{servers}|$ holds.

A seed changes the number of servers per site, the exact server capacity, and the distribution of all randomized processes (*e.g.*, actual network link latency per packet, actual number of servers per site, NFC chosen for a new flow arriving at an ingress). We run each benchmark with three seeds.

2) FUMES: We use an A3C [36] setup with: two shared layers (AT $\to$ 256 ; 256 $\to$ 256); two actor layers (256 $\to$ 128; 128 $\to$ AT); two critic layers (256 $\to$ 128; 128 $\to$ 1); all layers with rectified linear unit activation. The learning rate starts at 0.0001 and is reduced by 1% every 200 batches.

To highlight FUMES' ability to handle dynamism, we train it only once and use it for all benchmarks, although the benchmark setups differ from the training setup. One benchmark is an exception, where we show two instances of FUMES, FUMES-SND and FUMES-MDP, trained respectively on the SND+Abilene and MDP+Abilene setups, both with the NFCs of § V-A1, using three seeds. So FUMES-SND is trained on a different setup than what is used for the benchmarks, except in § V-C2. When we train FUMES, we fix the available server capacity to 115% and run for each seed $40,000$ batches, where one batch consists of $2,000$ tuples (scheduling decisions); see § V-B for more details. We set $\alpha = 0.9$, and $\beta = 0.5$.

3) Schedulers: We compare FUMES with these schedulers:

STEAM(++): An extension of STEAM [22] for supporting NFC branching. STEAM is a runtime packet-level scheduler that performs queueing predominantly at the NFCC (instead of at the NFI). This enables STEAM to perform scheduling decisions *later* compared to a scheduler like FUMES that performs queueing at the NFIs; this comes at the cost of high buffering demands at the NFCCs though (especially when the scheduler is saturated). Branching support is added by considering the first branch in any of the calculations when the decision is not known yet.

Greedy(-$\mathcal{O}$): Performs packet-level scheduling with queueing at NFIs. For each scheduling decision, Greedy calculates the estimated delay to serve the packet (sum of network delay, queuing and processing time at the NFI, and latency towards the egress), and greedily selects the fastest

option. Greedy has no mechanism to detect link latencies; it uses an *oracle* $\mathcal{O}$ *that has up-to-date distributions*.

4) Metrics: For each benchmark, we vary the available server capacity (abscissa) for the experiments, and report:

Success rate: Number of packets successfully delivered at the egress within the QoS limit of the chain. We focus on success rates for **server capacity** configurations in the most relevant and challenging range, where FUMES achieves values $\geq 90\%$. Also, we report required server capacity over target success rate for the range $90-100\%$.

Quality CDF: CDF of the scheduling quality $= 1-\text{PQ}$, with packets dropped at QoS timeouts capped with value 0.

Total reward: Accumulated total reward over training batches (for FUMES), yielding insights on convergence.

B. Training Performance (Q1)

We evaluate the convergence performance for FUMES-SND agents. When we train FUMES, we run three times $40,000$ batches, each using a different seed but running on the same setup with a fixed server capacity of 115%. Fig. 5a shows the achieved success rate over time. FUMES achieves already a success rate of over 90% during the first training iteration after $10,000$ batches. The second and third training iterations bring only little improvement on success rate, because FUMES is already in the 99%+ range. However, there is still improvement within a training iteration (the results of the iterations cannot be compared due to different seeds); success rate improves in the second iteration by $\approx 0.8\%$, and in the third by $\approx 0.3\%$.

C. Base Performance, Traffic Dynamism (Q2, [DYN-WORK])

We first use the Abilene topology with static NFC setup.

1) MDP traffic model: Fig. 7a shows the success rate over available server capacity, Fig. 7b focuses on the success rates $\geq 90\%$ and shows the required server capacity to reach these. FUMES-MDP shows close performance to FUMES-SND, requiring slightly more server capacity ($< 5\%$) to reach the same success rate, even though FUMES-MDP was trained on the benchmark workload. The MDP traffic's dynamism challenges training for FUMES-MDP. Greedy performs worst. STEAM outperforms the other schedulers up to a success rate of $\approx 96\%$. Above $\approx 96\%$ FUMES outperforms all schedulers.

2) SND traffic model: In Fig. 6 we change the traffic model and benchmark all schedulers on the SND traffic model, with both FUMES-MDP and FUMES-SND. Fig. 6a shows the success rate over available server capacity, normalized to 100% where FUMES achieves 90% success rate. FUMES-MDP shows close performance to FUMES-SND, requiring barely $\approx 5\%$ more server capacity to reach the same 90% success rate. The other schedulers show poor performance, not comparable with the performance we have seen in the previous experiments. One reason might be the choice of their hyperparameters, which were tuned for the setup of § V-C1 (because we use MDP in all following benchmarks).

Before we move to other benchmarks, we evaluate the schedulers' QoS rates. Fig. 6c shows the quality $(1-\text{PQ})$ for successfully delivered packets; higher is better. Fig. 6d shows one server capacity configuration, 115%, when all schedulers are still below 100% success rate. The analysis shows that FUMES prioritizes high success rates over improving QoS rates as long as success rate is below 100%. We focus on FUMES-SND in the following due to space constraints.

D. Site Topology Dynamism (Q2, [DYN-ENV])

We switch to the Nobel-Germany topology (see § V-A1) with MDP traffic workload. Again, we use FUMES-SND though to highlight dynamism. As the Abilene setup has 12 NFCCs vs 17 for the Nobel-Germany setup, when deploying the FUMES-SND on the Nobel-Germany setup, we randomly choose one trained agent and deploy it on all 17 Nobel-Germany NFCCs. This challenges FUMES' dynamic adaptation even more. Fig. 8 shows success rates. Again FUMES outperforms all other schedulers, especially for reaching higher percentages close to 100%. FUMES hits 99% with 20% lower server capacity, and the $99\%+$ with even more savings.

E. Unseen NFCs, Branching Dynamism (Q3, [DYN-WORK])

Next we change the NFC configuration as follows. We replace the previously used NFCs with 3 NFCs including branching: (NF_4), $(\text{NF}_4, [\text{NF}_2, \text{NF}_1]_{p=0.1} \vee [\text{NF}_1, \text{NF}_3]_{p=0.2} \vee [\text{NF}_3, \text{NF}_1, \text{NF}_5]_{p=0.7})$, and $(\text{NF}_1, \text{NF}_3, \text{NF}_4, \text{NF}_5, \text{NF}_2)$ with QoS deadlines $31ms$, $42ms$, and $48.5ms$ respectively. The second NFC has 3 possible branches with the highest probability of 0.7 for taking the longest branch. These experiments again use MDP traffic on the Nobel-Germany topology. Fig. 9 shows the success rates of the schedulers, which unveil a similar pattern as with the previous benchmark. FUMES-SND shows best performance, with even more saved server capacity to reach high success rates compared to STEAM.

F. Network Link Failures Dynamism (Q4, [DYN-ENV])

The last benchmark evaluates robustness in the event of link failure. This benchmark extends the previous setup § V-E as follows. At any point in time one NFCC-NFCC link is turned to failure mode, multiplying its link latency distribution values with $1,000$ compared to normal mode. A failed link returns to normal operation mode after $50s$ and the next link is randomly chosen to be set to failure mode. Fig. 10 shows the success rate over time, each with a setup where link failures are injected (dashed) and without (solid). Clearly FUMES is not affected by link failures. STEAM, conversely, shows a clear performance drop with link failures, with success rate dropping by up to $\approx 20\%$ w.r.t. its performance without failures. Greedy has the bonus of being given an oracle for link latencies, knowing exactly when a link goes to failure mode. Still it shows overall bad success rates.

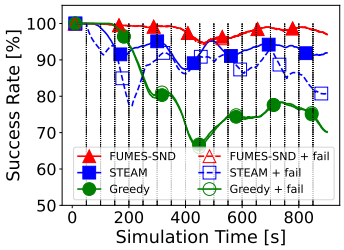
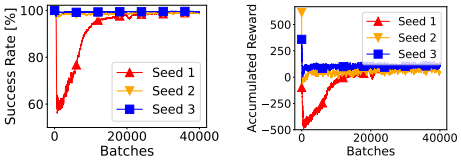
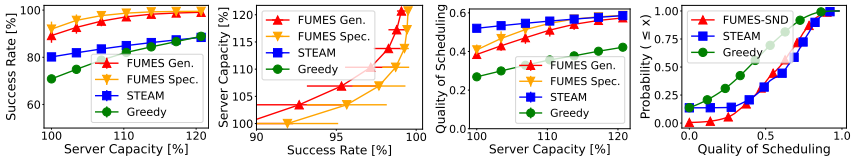


Fig. 10: FUMES robustness to link failures: success rate. Each vertical line shows a different link being set to failed mode.



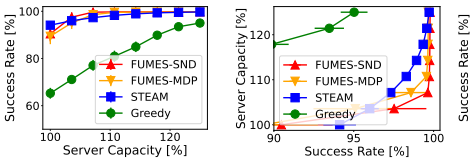
(a) Success rate over batches (b) Total reward over batches

Fig. 5: Training convergence of FUMES.



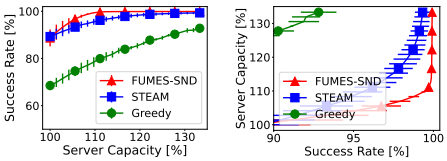
(a) Success rate vs capacity (b) Capacity vs success rate (c) Quality vs capacity (d) Quality CDF

Fig. 6: SND workload, Abilene topology, static NFCs.



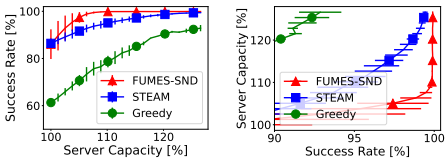
(a) Success rate vs server capacity (b) Req. server cap. vs success rate

Fig. 7: MDP workload, Abilene topology, static NFCs.



(a) Success rate vs capacity (b) Capacity vs success rate

Fig. 8: MDP workload, Nobel-Germany topology, static NFCs.



(a) Success rate vs capacity (b) Capacity vs success rate

Fig. 9: MDP workload, Nobel-Germany, branching NFCs.

VI. Related Work

Among the many works on VNFs or NFCs, several attempt to support dynamism in workloads ([DYN-WORK]) by periodically adjusting *deployment* of NFIs or VNFs and assignment of resources such as CPU cores to each of them. Several works consider optimizations and scheduling at the level of a *single server or CPU core* only. *E.g.*, NFVnice [37] is a VNF framework for CPUs that aims for fair and efficient resource allocation of chains, considering the impact of different VNFs on resource usage. Katsikas et al. [38] allow to reduce inter-core transfers of packets on the server and by this improve single-server VNFs throughput. Meng et al. [39] split an NFC into smaller VNFs, retaining semantics, enabling reuse of parts of an NFC across others. Satyam et al. [7] study VNF placement and CPU allocation for *co-located* VNFs to minimize latency, however assuming a priori knowledge of packet arrival rates. Others involve disruptive NFI migration [6]. None can react in real-time to workload fluctuations across networks.

Solutions for *scheduling* packets for individual VNFs or NFCs mostly perform network-wide centralized scheduling, assuming global network state knowledge and advance knowledge of statistical information – often even assuming static bandwidth for flows. *E.g.*, Mechtri et al. [11] consider joint NFC placement and scheduling for systems forming undirected graphs, using *a priori* knowledge of the static bandwidth of each flow. Others consider similar setups [12]–[17].

Few solutions consider dynamic flow demands. Qu et al. [5] limit servers to running single NF instances and links to transferring single flows at a time. Eramo et al. [6] allow traffic to change, but still require knowledge of flows' nominal and maximum traffic volumes. Anwer et al. [9] use the input and output traffic volume of all NFIs to dynamically update the routing of NFCs. Bhamara et al. [40] use queuing models for servers and links in a multi-cloud environment to minimize cross-cloud traffic and response times but assume a priori knowledge of packet arrival rates. Other solutions cannot be applied work in real-time due to prohibitively high complexity [8], with seconds to take effect [10].

Several recent works leverage machine learning techniques in the context of NFC routing. Pei et al. [18] use neural networks to select NFIs in a way minimizing latency. Ning et al. [19] leverage RL to miminize latency and also balance load and avoid bad links. Schneider et al. [20] also use deep RL for NFI scheduling using periodically updated scheduling tables; while every node has its tables, these contain state of the entire network, and are updated by a centralized controller, as used also in the prior two works. Lin et al. [21] use RL for both NFI placement and scheduling, however requiring flow rate estimates, focusing on activating/deactivating NFIs as measure, and optimizing for costs not including QoS.

The closest work to FUMES is STEAM [22], the first to treat traffic scheduling fully as runtime problem without global oracles, centralized components, or a priori knowledge of traffic patterns. As shown, STEAM supports neither expressive NFCs with branching (needed for [DYN-WORK]), nor environmental changes like link failures ([DYN-ENV]).

VII. Conclusions

We proposed FUMES for runtime NFC scheduling in real-life dynamic settings. FUMES leverages a MARL technique for distributed scheduling to meet strict QoS requirements across NFCs. We discussed challenges and proposed efficient solutions to address them in our learning-based approach. Experimental results with both synthetic and real-life network topologies and traffic traces show that FUMES achieves significantly better packet delivery success rates while reducing required server capacity w.r.t. existing approaches. Moreover, FUMES can adapt to different network setups and conditions without expensive retraining. We see further avenues in moving agent inference out of the critical path to minimize latency overhead of scheduling, and in considering stateful NFIs.

Acknowledgements

This work has been funded by the Federal Ministry of Education and Research (BMBF) Software Campus grant 01IS17050 “dynSFC”, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 210487104 – SFB 1053, the Hasler Foundation, and the Swiss National Science Foundation grant 192121 “FORWARD”.

References

[1] IETF, “Service Function Chaining Use Cases in Mobile Networks,” Tech. Rep. draft-ietf-sfc-use-case-mobility-09, Jan. 2019.

[2] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in *SOSE*, 2019.

[3] K. Kaur, V. Mangat, and K. Kumar, “A comprehensive survey of service function chain provisioning approaches in SDN and NFV architecture,” *Computer Science Review*, vol. 38, p. 100298, 2020.

[4] C. J. Bernardos and A. Mourad, “Distributed SFC control for fog environments,” IETF, Internet-Draft draft-bernardos-sfc-distributed-control-06, Sep. 2022, Work in progress.

[5] L. Qu, C. Assi, and K. Shaban, “Delay-aware scheduling and resource optimization with network function virtualization,” *IEEE TCOM*, vol. 64, no. 9, pp. 3746–3758, 2016.

[6] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, “An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures,” *IEEE/ACM TON*, vol. 25, no. 4, pp. 2008–2025, 2017.

[7] A. Satyam, M. Francesco, C. F. Chiasserini, and D. Swedes, “Joint VNF Placement and CPU Allocation in 5G,” in *INFOCOM*, 2018.

[8] Q. Zhang, F. Liu, and C. Zeng, “Adaptive interference-aware VNF placement for service-customized 5G network slices,” in *INFOCOM*, 2019.

[9] B. Anwer, T. Benson, N. Feamster, and D. Levin, “Programming Slick Network Functions,” in *SOSR*, 2015.

[10] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: a framework for nfv applications,” in *SOSP*, 2015.

[11] M. Mechtri, C. Ghribi, and D. Zeghlache, “A scalable algorithm for the placement of service function chains,” *IEEE TNSM*, vol. 13, no. 3, pp. 533–546, 2016.

[12] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, “Near optimal placement of virtual network functions,” in *INFOCOM*, 2015.

[13] B. Addis, D. Belabed, M. Bouet, and S. Secci, “Virtual network functions placement and routing optimization,” in *CloudNet*, 2015.

[14] B. Martini, F. Paganelli, P. Cappanera, S. Turchi, and P. Castoldi, “Latency-aware composition of virtual functions in 5G,” in *NetSoft*, 2015.

[15] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, “Design and evaluation of algorithms for mapping and scheduling of virtual network functions,” in *NetSoft*, 2015.

[16] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta, “Joint optimization of service function chaining and resource allocation in network function virtualization,” *IEEE Access*, vol. 4, pp. 8084–8094, 2016.

[17] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, “Deploying chains of virtual network functions: On the relation between link and server usage,” in *INFOCOM*, 2016.

[18] J. Pei, P. Hong, K. Xue, D. Li, D. S. L. Wei, and F. Wu, “Two-phase virtual network function selection and chaining algorithm based on deep learning in sdn/nfv-enabled networks,” *IEEE JSAC*, vol. 38, no. 6, pp. 1102–1117, 2020.

[19] Z. Ning, N. Wang, and R. Tafazolli, “Deep reinforcement learning for nfv-based service function chaining in multi-service networks: invited paper,” in *HPSR*, 2020.

[20] S. Schneider, R. Khalili, A. Manzoor, H. Qarawlus, R. Schellenberg, H. Karl, and A. Hecker, “Self-learning multi-objective service coordination using deep reinforcement learning,” *IEEE TNSM*, vol. 18, no. 3, pp. 3829–3842, 2021.

[21] L. Gu, D. Zeng, W. Li, S. Guo, A. Y. Zomaya, and H. Jin, “Intelligent VNF orchestration and flow scheduling via model-assisted deep reinforcement learning,” *IEEE JSAC*, vol. 38, no. 2, pp. 279–291, 2020.

[22] M. Blöcher, R. Khalili, L. Wang, and P. Eugster, “Letting off STEAM: distributed runtime traffic scheduling for service function chaining,” in *INFOCOM*, 2020.

[23] L. S. Shapley, “Stochastic games*,” *PNAS*, vol. 39, no. 10, pp. 1095–1100, 1953.

[24] J. M. Halpern and C. Pignataro, “Service Function Chaining (SFC) Architecture,” RFC 7665, 2015.

[25] T. L. Foundation, “Kubernetes,” https://kubernetes.io.

[26] Cisco, “Best Practices in Core Network Capacity Planning,” Tech. Rep.

[27] R. Nadiv and T. Naveh, “Wireless backhaul topologies: Analyzing backhaul topology strategies,” *White Paper Ceragon*, 2010.

[28] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[29] M. Kablan, A. Alsudais, E. Keller, and F. Le, “Stateless network functions: Breaking the tight coupling of state and processing,” in *NSDI*, 2017.

[30] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, “Let it flow: Resilient asymmetric load balancing with flowlet switching,” *NSDI*, 2017.

[31] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *CoRR*, vol. cs.AI/9605103, 1996. [Online]. Available: https://arxiv.org/abs/cs/9605103

[32] D. Mills, J. Martin, J. Burbank, and W. Kasch, “Network Time Protocol Version 4: Protocol and Algorithms Specification,” RFC 5905, 2010.

[33] S. Mehraghdam, M. Keller, and H. Karl, “Specifying and placing chains of virtual network functions,” in *CloudNet*, 2014.

[34] Q. Zhang, Y. Xiao, F. Liu, J. C. Lui, J. Guo, and T. Wang, “Joint optimization of chain placement and request scheduling for network function virtualization,” in *ICDCS*, 2017.

[35] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, “SNDlib 1.0–Survivable Network Design Library,” *Networks*, vol. 55, no. 3, pp. 276–286, 2010. [Online]. Available: http://sndlib.zib.de

[36] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*. PMLR, 2016.

[37] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, “NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains,” in *SIGCOMM*, 2017.

[38] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., “Metron: NFV Service Chains at the True Speed of the Underlying Hardware,” in *NSDI*, 2018.

[39] Z. Meng, J. Bi, C. Sun, H. Wang, and H. Hu, “CoCo: Compact and Optimized Consolidation of Modularized Service Function Chains in NFV,” in *ICC*, 2018.

[40] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, “Optimal virtual network function placement in multi-cloud service function chaining architecture,” *COMCOM*, vol. 102, pp. 1–16, 2017.