DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-level Visibility

Qingqing Yang, Xi Peng, Li Chen*, Libin Liu#, Jingze Zhang†, Hong Xu†, Baochun Li‡, Gong Zhang

Huawei Theory Lab, *Zhongguancun Laboratory, †Chinese University of Hong Kong,
#Shandong Computer Science Center (National Supercomputer Center in Jinan), ‡University of Toronto

ABSTRACT

Network simulators are an essential tool for network operators, and can assist important tasks such as capacity planning, topology design, and parameter tuning. Popular simulators are all based on discrete event simulation, and their performance does not scale with the size of modern networks. Recently, deep-learning-based techniques are introduced to solve the scalability problem, but, as we show with experiments, they have poor visibility in their simulation results, and cannot generalize to diverse scenarios. In this work, we combine scalable and generalized continuous simulation techniques with discrete event simulation to achieve high scalability, while providing packet-level visibility. We start from a solid queueing-theoretic modeling of modern networks, and carefully identify the mathematically-intractable or computationally-expensive parts, only which are then modeled using deep neural networks (DNN). Dubbed *DeepQueueNet*, our approach combines prior knowledge of networks, and supports arbitrary topology and device traffic management mechanisms (given sufficient training data). Our extensive experiments show that DeepQueueNet achieves near-linear speedup in the number of GPUs, and its estimation accuracy for average and 99th percentile round-trip time outperforms existing end-to-end DNN-based performance estimators in all scenarios.

CCS CONCEPTS

• **Networks** → **Network performance modeling**; **Network simulations**; *Network experimentation*; • **Computing methodologies** → *Massively parallel and high-performance simulations*;

KEYWORDS

Network simulation, Network performance estimation, Machine learning, Queueing theory, Network modeling

ACM Reference Format:
Qingqing Yang, Xi Peng, Li Chen*, Libin Liu#, Jingze Zhang†, Hong Xu†, Baochun Li‡, Gong Zhang. 2022. DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-level Visibility. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3544216.3544248

Corresponding Author: Li Chen (lichen@zgclab.edu.cn).

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544248

1 INTRODUCTION

Network simulators are an essential tool for network operators, and can assist in important tasks such as capacity planning, topology design, and parameter tuning. Conventional network simulators are based on discrete event simulation (DES) [20, 35, 39, 46], but as modern networks grow rapidly in their scales, the performance of DES simulators fails to keep up. The fundamental obstacle is the dependency between discrete events, and even distributed and parallelized implementations of DES cannot increase its performance satisfactorily [22, 39].

To make network simulation more scalable, recent efforts have sought to build end-to-end performance estimators (EPEs) using machine learning and continuous simulation [26, 41, 42], rather than packet-by-packet DES. An EPE encapsulates the network as a deep neural network (DNN) model, which can be trained in an end-to-end fashion using real traffic traces. Given an embedding of facts of the network, EPE directly predicts end-to-end performance metrics such as the round-trip time (RTT), flow completion time (FCT), and packet drop rate. This has the obvious performance advantage because the DNN models can easily be parallelized in both the training and inference stages. Although they can achieve impressive scalability, their usefulness is still questionable:

- **Packet-level Visibility**: Current EPEs fail to answer questions about specific device or flow, and cannot give packet-level statistics (more details in § 2.3).
- **Generality**: Current EPEs cannot provide reliable and accurate estimates when the network configuration (topology, traffic generation models, and device traffic management (TM) mechanisms) is changed (more details in § 2.3). It is not economical to re-collect traffic traces of the entire network and re-train the model just to accommodate a minor change in simulation settings.

Recently, to enhance visibility and generality of EPEs, Zhang et al. [23, 50] propose to combine DES and EPE with MimicNet. MimicNet adopts DES to quickly generate accurate data for a subset (cluster) of the larger network, then uses the data to train a "mimic"–a DNN-based estimator of a cluster's performance, and finally composes the mimics to form simulations of complete datacenter networks. However, MimicNet only works for FatTree [30], and focuses on scale generalizability.

What we learn from prior efforts is that, to achieve scalability that suits the scale of modern networks, DNNs are an indispensable

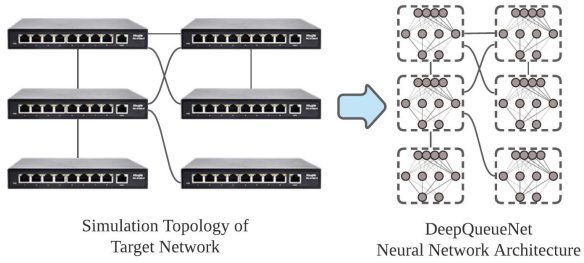


Figure 1: DeepQueueNet: Mapping a target topology directly to a neural network architecture.

tool to approximate the unknown and mathematically intractable performance function of modern large-scale networks; but the main problem is the generality of DNN-based solutions. MimicNet sheds the first light on conquering this problem by reducing the scope of the DNN model from network-scale to cluster-scale, and in this paper, we seek to understand the following question: can we further improve generality of network performance estimation if we narrow the scope of DNN down to the smallest possible scale?

To this end, we embrace the continuous simulation paradigm, which is popular for weather and physics simulations [13, 17, 19], as well as flow-level simulation for transport layer dynamics in networking research [2, 31, 34, 37]. Continuous simulations start by describing the underlying mechanisms of the target system using mathematical equations of system states (e.g. partial differential equations (PDE)), and they compute the evolution of system state. We believe the fundamental pitfall of prior work is that they attempt to directly model the performance function of a group of devices as a black-box using DNNs and forego the essential step of describing the behavior of the target system with precise mathematical formulations.

In this work, we aim to build a scalable and general network performance estimator with packet-level visibility. In contrast to prior efforts, our approach combines DES and continuous simulation. At a high-level, our proposed methodology is as follows: we first build a theoretical foundation to express our prior knowledge of the network as much as possible, then identify the parts that are mathematically intractable or computationally expensive, and finally replace these parts with DNNs that can be trained using real data traces.

Following this methodology, our main contribution is towards narrowing down the scope of the application of DNNs in EPE—from network/cluster-scale to device-scale. We use DNNs *only* to model device-local TM mechanisms, which is shown to be computationally expensive by our queueing-theoretic modeling and numerical simulations (§ 2.2). Specifically, we model the packet stream arriving at each of the ingress ports of the device as a time series, and the device model processes the set of ingress time series to produce a set of egress time series, which then are fed to the down-stream devices as their ingress time series. We compose the device model using two sub-models: a packet-level forwarding model (PFM) and a packet-level TM model (PTM). The PFM specifies the forwarding behavior, which can be described explicitly using tensor multiplication given the routing table. The PTM predicts how much delay is experienced for each packet (the latency of dropped packet is $+\infty$). Each PTM is a DNN with a customized Transformer [9] architecture which we find to be general enough to encapsulate both traffic variations and TM differences. Training data for PTM is relatively easy to generate, because we only need to collect device-local ingress and egress packet traces. With trained device models, we can connect them with links[1] to setup arbitrary network topology. In this way, the simulation topology is mapped directly to DeepQueueNet's neural network architecture (Figure 1).

The workflow of DeepQueueNet is the same as existing DES implementations. For both DES and DeepQueueNet, we start by constructing the network topology with device models in the simulator, prepare input packet traces and run the trace on the simulator, collect output traces, and summarize the output trace using different metrics. Due to DeepQueueNet's packet-level visibility, any new metric can be applied to the output trace without retraining. The key difference is that, while DES obtains per-packet latency on each device sequentially, DeepQueueNet predicts packet latencies in batches by model inference. We release an open source demo of DeepQueueNet to illustrate this workflow[2].

We summarize our contributions as follows:

- **Packet-level Visibility**: DeepQueueNet models the traffic flow as a time series of packets, and the devices as "operators" on the time series. By this construction, users of DeepQueueNet can obtain final packet traces for each device, which helps them to understand the final performance metric and identify problematic device or flows. This helps to answer important questions such as which device introduces the most delay to a flow, or where is the location of the bottleneck of the topology given a traffic pattern.
- **Generality**: With the expressive device-local PTM models, DeepQueueNet is also generalizable for arbitrary network topology, traffic generation models, and device-local TM mechanisms. We develop two algorithms, SEC and IRSA. SEC mitigate error propagation and improve accuracy, while IRSA maintains the correctness of time order between packets across all devices via an iterative approach (§ 4). As our experiments show (§ 6), in all scenarios with different configurations, DeepQueueNet can achieve superior accuracy for average and 99th percentile per-packet round-trip time (RTT) and jitter compared to existing EPEs.
- **Scalability**: DeepQueueNet is scalable due to two factors: 1) As an neural network, inference of DeepQueueNet can be accelerated in parallel easily with current distributed deep learning frameworks; 2) DeepQueueNet process packets in batches, and to mitigate cross-batch packet reordering, we design an iterative re-sequencing algorithm and prove its convergence. Our current prototype supports multi-GPU training and inference; it is able to achieve near-linear speed-up with the number of GPUs (§ 6).

Caveat: Although DeepQueueNet achieves significant acceleration, it ignores the state-ful interactions among the upper layer applications, the transport protocols they uses, and the network-scale dynamic traffic steering. In essence, DeepQueueNet is a sequence-to-sequence predictor for general networks, and we infer the performance of the target network by examining the latencies experienced

[1] Links are also devices: a link model is a device model with a pair of input and output ports.
[2] https://github.com/HUAWEI-Theory-Lab/deepqueuenet

by all the packets on each link of the simulation. The current prototype of DeepQueueNet only looks at information at the network layer, *i.e.,* the path of a packet takes, the size of the packet, the inter-arrival time of the packet, and the arrival and departure time of the packet at each device on its path. DeepQueueNet helps users to understand the following question: given a packet trace, a network topology, and the TM configuration of the devices in the network, what will the output packet trace be for each link in the topology? We intentionally make the trade-off favoring performance over accuracy, because, in our experience with network operators and architects, for important network-scale tasks like capacity planning, topology design, and parameter tuning, they are often more interested in the performance distribution when a network reaches stability. DeepQueueNet is appropriate for these scenarios.

In the following, we first overview the background and motivate the design of DeepQueueNet in § 2, and present its system design in § 3. We examine the techniques and algorithms in § 4 and evaluate our prototype implementation in § 6.

Disclaimer: This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

In this section, we overview three types of network simulation systems: packet-level DES, flow-level continuous simulators, and recent DNN-based EPEs. After understanding their benefits and limitations, we motivate the design of DeepQueueNet.

2.1 Packet-level DES

There is a variety of DES software, such as NS2/3 [21, 39], OMNeT++ [46] and OPNET [29]. Packet-level DES is the most explainable type of simulation, as the entities in simulator have direct correspondence to the target network, and we can collect the packet traces on each device/link to gain deep understanding of the simulation. Because DES processes each packet sequentially, any TM mechanisms can be specified.

However, DES has scalability issue when the size of the simulation increases, and much effort has been made to build parallel and distributed DES (PDDES), which employs multiprocessor and distributed computing platforms. However, due to messaging and synchronization overhead between processes and machines, PDDES is *not* guaranteed to provide any speedup. In bad cases, PDDES might execute significantly slower than sequential simulation [39]. Also, the performance and scalability of PDDES is constrained by fine-grain communication, especially in execution environments with high communication cost [22], and the reliability and scalability remain problematic.

2.2 Flow-level Continuous Simulators

These type simulators are based on the abstraction of traffic flows and target network, and there are three branches of related work:

Control-theoretic estimators are often used in the design of transport layer traffic control protocols [2, 31, 34, 37]. These simulators require users to specify PDEs of the system states and interaction dynamics [2, 4]. Although there are scalable PDE solvers that can accelerate this type of simulators, their usefulness is limited. Firstly, they require predefined PDEs of flow-level dynamics for all flows, which requires expertise and is often unavailable. Secondly, for mathematical convenience, they often use fluid model that assumes the packet size to be arbitrarily small, therefore they are too simplistic to reflect real traffic and network devices TM mechanisms. Most importantly, with only solutions to steady states, they cannot produce useful statistics such as distribution of latency, which is arguably most important for practical network engineering.

Network calculus (NC) [8, 25] approximates the envelopes of flow-level arrivals and services by using wide-sense increasing functions, rather than exact values. Using min-plus and max-plus algebras, NC is a theoretical framework for analyzing the worst-case bounds on delay and backlog of a network. Although NC can handle a network of schedulers, it gives only loose bounds, and cannot provide estimates for average or any other percentile, limiting its usefulness in practice. For accurate latency distribution estimations, we must turn to queueing theory.

Queueing-theoretic estimators: Queueing theory [40] enables the evaluation of network performance by describing the system behavior with a queueing model which are composed of the modeling of packet arrivals and the modeling of scheduling servers. The packet-level queueing model can approximate latency and loss very accurately at the expense of high complexity. For example, the Markovian arrival process (MAP) [3, 6] and its extensions have superior representational capacity for the stochastic processes of packet arrivals and services. Although they are analytically tractable based on continuous-time Markov chain (CTMC) and thus allow the analysis of latency distribution and queue-length distribution, this type of models has to hold a large state space which has a high computational complexity. Due to space limits and the fear of confusing readers with different definition and formulation of traffic flows and schedulers, we elaborate our queueing theoretic results in the Appendix for interested readers. In the Appendix, we show how an accurate MAP-based modeling applies for real network traffic. Moreover, we also make the first attempt to analyze a general multi-queue scheduler (deficit round robin (DRR), weighted fair queueing (WFQ), weighted round robin (WRR), and strict priority (SP)) by extending the packetized general processor sharing (PGPS) method, and our resultant queueing-theoretic estimator achieves high accuracy in terms of latency estimation. However, we also emphasize that the time complexity is exponential with respect to both the number of queues and the size of buffer (see Appendix B.2).

Summary: The main take-away is that high-accuracy queueing-theoretic estimator are computationally expensive and infeasible. Despite our best efforts, the accurate queueing-theoretic model can hardly scale beyond a single device.

2.3 End-to-end Performance Estimators

The research community have shown great interest in building DNN-based EPEs due to their scalability with respect to the size of the target system. Most prior efforts focus on applying advanced DNN architectures from other domains, such as computer vision [49] and graph learning [48], and they attempt to use DNN to model the performance metrics of a network in an end-to-end manner. SimNet [26] targets computer architecture simulation and uses convolutional neural networks. RouteNet [41, 42] employs graph neural networks to predict key performance indicators of a network, such as delay, jitter, and packet loss rate. DNN allows these EPE systems

to achieve high scalability due to highly efficient, parallel DNN inference infrastructure developed in the past decade [1, 36, 45]. However, they have the following deficiencies:

- **Packet-level Visibility**: These EPEs fail to answer questions about specific devices or flows. They learn a mapping from the embedding of network facts to predetermined performance metrics. There lacks connections between the final metrics and traffic characteristics, topology, or device configurations. In addition, when the user becomes interested in a new metric, training dataset must be recollected, and the model redesigned and retrained.
- **Generality**: These EPEs cannot provide reliable and accurate estimates when the network configuration is changed. It is uneconomical to recollect traffic traces and retrain the model, just to accommodate a minor change in simulation setting:
  - **Network topology**: When the topology or the network's size changes, these EPEs must be retrained.
  - **Traffic patterns**: We show that, by varying flow generation parameters, existing EPEs' performance metric can differ greatly from that of a DES (§ 6.1). This means when traffic pattern changes, current EPEs must be retrained.
  - **Traffic management (TM)**: Traffic management mechanisms have huge impact on network performance. Current EPEs assumes FIFO queueing discipline, while in practice, diverse and complicated scheduling mechanisms are all used: WRR, DRR, WFQ, SP, *etc.* Existing EPEs are not designed to support different schedulers. Besides packet scheduling, current EPEs also cannot support other popular TM mechanisms such as buffer management and dynamic load balancing.

A recent system, MimicNet [23, 50], overcomes most of these deficiencies by combining DES and EPE. MimicNet uses DES to perform packet-level accurate simulation for a subset (cluster) of a datacenter network, which includes detailed queueing and transport layer dynamics. It then uses the collected data from this observable subset of network to train "mimic"s, which are approximators of the non-observable internal and cross-cluster behavior. Finally, they compose the observable cluster with the mimic clusters to form a full-scale datacenter network simulation. Compared to prior work, this approach achieves orders of magnitude reduction in simulation completion time, with moderate loss in metric accuracy. We believe the gains of MimicNet are, in part, due to narrowing the scope of DNN from network-scale to cluster-scale. However, in its effort to achieve scale-generality, MimicNet targets the FatTree [30] topology, and it cannot adapt to other network topologies.

2.4 Motivation and Design Rationale

Our design goal for DeepQueueNet is to combine the advantages of all three types of network simulators: *visible and general like DES, accurate like single-hop queueing-theoretic models, and scalable like DNN-based EPE.* At the same time, we intend to avoid their drawbacks.

We believe the key to visibility and generality of DES is that it allows users to examine per-device traffic traces both before and after entering the device. We thus decide to model each device as an "operator" that transforms ingress packet streams to egress packet streams. For performance simulation, each device has two functionalities: forwarding and TM mechanism. The forwarding part of this operator can be straightforwardly abstracted as a tensor multiplication (§ 3.2.2). The challenge lies in the TM part. As we analyzed above, to the best of our knowledge, the most accurate model is a queueing-theoretic one, but its exponential complexity is computationally prohibitive. Thus we instead use DNN to model this part of the operator (§ 3.2.2), which has constant time-complexity for model inference. Compared to MimicNet that narrows down the scope of DNN from network-scale to device-scale, our approach further contracts the scope to only the TM part of a device. We train this per-device DNN using ingress and egress packet traces collected from devices. Trained model can be used to build arbitrary topology, and the output traces of each device is simply a packet trace, on which any new metric can be applied without retraining the model.

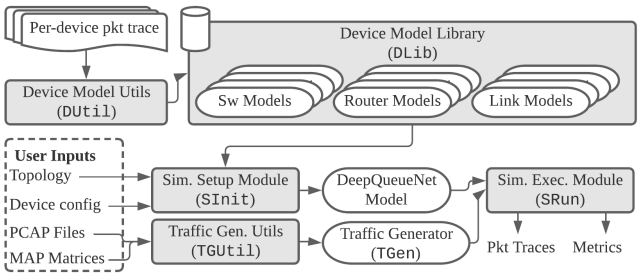


Figure 2: DeepQueueNet System Architecture

This DNN-based device model is intrinsically scalable with respect to simulation size. Each device model in DeepQueueNet processes the ingress packet stream in batches, and DeepQueueNet itself can be parallelized to multiple machines and GPUs using distributed deep learning frameworks.

The key challenge for DeepQueueNet is ensuring generality for topologies, traffic patterns, and TM mechanisms. We choose the expressive Transformer DNN architecture [9], and design feature engineering, iterative re-sequencing (IRSA), and statistical error correction (SEC), which are elaborated in § 4.

Assumptions: In the following, we assume the routing table and the flow-to-priority/weight assignments for packet schedulers is given in the setup phase of simulation and stable during the simulation. This is because we design DeepQueueNet to be a network performance simulation system for large-scale networks. Building from a queueing-theoretic foundation, DeepQueueNet focuses on modeling the performance distribution of a system when it reaches stability, and in general ignores the transient behaviors of the system. DeepQueueNet also processes packets in batches to reach high scalability, ignoring possible interactions in the process. Therefore, it is *not* suitable for detailed, interaction-oriented, and state-ful protocol simulations, *e.g.*, transport layer protocols or routing protocols.

3 DEEPQUEUENET DESIGN

In this section, we first overview the system architecture and workflow of DeepQueueNet, then we present the core models of DeepQueueNet: packet stream, device, and network.

3.1 System Overview

3.1.1 System Components The architecture of DeepQueueNet is shown in Figure 2, consisting of five core components:

Device Model Utilities (`DUtil`) produces trained device models. Given ingress and egress packet traces of a device, it trains a device model, and stores it in Device Model Library for construction of simulation topologies.

Device Model Library (`DLib`) stores and indexes trained device models, including switches, routers, and links.

Traffic Generation Utilities (`TGUtil`) creates traffic generators (`TGen`) based on user specification. In the simulation setup phase, `TGens` produces ingress packet streams, and batches them into ingress tensors. Currently, to customize `TGens`, users can use an existing set of PCAP files [33], or provide a set of MAP matrices of flows. This module can also extend to other traffic generation methods using the same traffic generator interface.

Simulation Setup Module (`SInit`) parses user input and setup the simulations. User input should contain the topology of the network, configuration of each node (device type and routing table), and traffic generators. It obtains specified device models from `DLib`, and composes them based on the topology into a DeepQueueNet model. It also uses `TGen` to generate ingress tensors for the DeepQueueNet model to inference. Finally, based on the routing tables, `SInit` creates tensors that describes the packet forwarding behavior of each device.

Simulation Execution Module (`SRun`) runs the simulations. It interfaces with the underlying deep learning framework and performs model inference. It uses the iterative re-sequencing algorithm (§ 3.2.4) and statistical error correction algorithm to refine the inference results. In the end, it outputs the final packet traces (time series) of each device in the simulation topology.

3.1.2 Workflow DeepQueueNet's workflow is the same as that of existing DES implementations: prepare simulation settings (topology, device configuration, and traffic generators), run the simulation, collect packet traces, and analyse by applying arbitrary metric to the results.

We highlight two key differences versus DES:

- **Specifying device model:** In DES, the device behaviors are specified by coding its packet processing mechanisms, which requires significant human effort. In contrast, DeepQueueNet specifies the device's packet processing behavior in a data-driven manner. For TM mechanisms, we train a black-box DNN model using real ingress and egress packet traces of the device. For packet forwarding, we abstract it as a tensor multiplication to guarantee correctness (more details in § 3.2.2).
- **Processing packets:** DES obtains per-packet latency on each device sequentially, while DeepQueueNet predicts packet latencies in batches by model inference, which is more scalable with respect to simulation size.

Next, we describe the modeling of packet streams, devices, and networks.

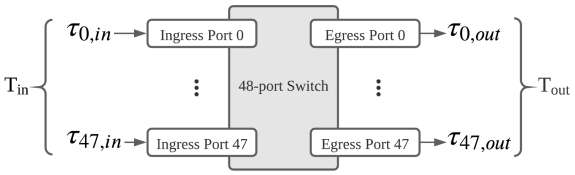


Figure 3: Example: Device model's input & output (of a 48-port Switch)

3.2 DeepQueueNet Modeling

3.2.1 Modeling Packet Streams We model a packet as a feature vector containing its information. In the current implementation, we use the following features: unique identification (ID) number (pid), flow ID (fid), packet length (len), transport protocol (trp). The packet vector p is thus:

$$p=<\text{pid,fid,len,trp}> \tag{1}$$

This vector can also be extended to include any additional features, such as applications or virtual network ID, if future dataset suggests that they improve the quality of latency predication. For our current dataset, we believe the above four features are sufficient.

A packet stream, τ, is a time series of packet arrivals. For a packet stream of n packets:

$$\tau=[(p_0,t_0),(p_1,t_1),(p_2,t_2),\cdots,(p_n,t_n)], where\ t_i \le t_{i+1}, \forall i \ge 0 \tag{2}$$

t_i is the arrival time of the i^{th} packet in the stream τ.

A network device, *e.g.*, a router, usually has more than one port. We split the physical port into a logical ingress port and a logical egress port, as shown in Figure 3. We model the collection of all ingress packets streams to a device as T_{in}. For a network device with k ports:

$$T_{in}=[\tau_{0,in};\tau_{1,in};\cdots;\tau_{k-1,in}] \tag{3}$$

$\tau_{j,in}$ is the ingress packet stream of the j^{th} port of the device. Likewise, the collection of egress streams is:

$$T_{out}=[\tau_{0,out};\tau_{1,out};\cdots;\tau_{k-1,out}] \tag{4}$$

3.2.2 Modeling Devices We start with the link model as an introductory example, because links are the simplest type of device with only one input and one output port. Assuming the length of the link is l, the propagation speed on the link c, and the bandwidth C, the relationship between the ingress and egress stream of the link is thus:

$$\begin{aligned}\tau_{out}&=\tau_{in}+[\vec{0},len(\tau_{in})/C+l/c]\\&=[\cdots,(p_{i,in},t_i+len(p_{i,in})/C+l/c),\cdots]\end{aligned} \tag{5}$$

$len(\cdot)$ is a pure function which can be mapped to τ to extract the packet lengths, and the zero vector $(\vec{0})$ indicates that the link does not modify packet vectors. In other words, a link device is an operator that adds a latency to all packets in the ingress time series, based on each packet's length and the setting of the link (l,c, and C).

Generally, for a multi-port network device, its device model has two sub-models: packet-level forwarding model (PFM) and packet-level TM model (PTM). We explain them in details below.

Packet-level Forwarding Model. The packet-level forwarding model specifies the forwarding behavior, which can be described explicitly using tensor multiplication given the forwarding table.

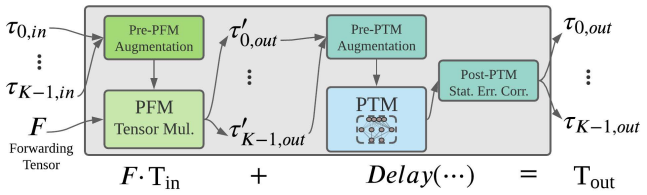


Figure 4: DeepQueueNet Device Model (K-port)

Before forwarding, we first augment the packet stream of each ingress port by adding the ingress port ID as a new feature in the packet vectors, which becomes <pid,fid,len,trp,in_port>. For all the ingress streams, we pad them to the same length with empty packets.

We assume the forwarding table of each device is generated or given before the start of the simulation, and we model it as function that takes the flow ID of the packet and the ID of the ingress port, and outputs the egress port ID.

$$forward(\text{fid,in\_port}) \rightarrow \text{out\_port} \tag{6}$$

With the augmented ingress streams and the $forward(\cdot)$ function, we can produce the forwarding tensor, F, which is a 3-dimensional 0-1 tensor. F=$[f_{i,j,k}]$, where $f_{i,j,k} \in 0,1$. F is of shape $K \times K \times N$, where K is the device's number of ports and N is the number of packets in the ingress streams. If $f_{i,j,k}$=1, it indicates that the k^{th} packet in the ingress stream of port i is forwarded to the egress port j. Otherwise, $f_{i,j,k}$=0.

Denote the collections of ingress and egress streams of the device as $\mathrm{T_{in}}$ and $\mathrm{T_{out}}$, respectively, and we have:

$$\mathrm{T_{out}} = \mathrm{F} \cdot \mathrm{T_{in}} + \mathrm{Delay}(\cdots) \tag{7}$$

In essence, the above tensor multiplication mixes the ingress packet streams to form egress packets streams, based on the device's forwarding table. The key point here is that, using Equation 7, we can process the forwarding in batches to enable high scalability, while DES can only process each packet sequentially.

The $Delay(\cdot)$ function in Equation 7 is PTM which we discuss next.

Packet-level TM Model. A PTM model predicts how much delay (sojourn time) is experienced for each packet. Like PFM, it processes the packets in batches, and adds a latency to each packet in $\mathrm{T_{out}}$. PTM starts by augmenting the packet vectors with features related to TM mechanism. We elaborate on the data augmentation method in § 4.1. With the ingress packets streams prepared, the PTM process them to add delays to each packet by performing inference of pre-trained DNN models. We refine the resultant egress packets streams with a post-PTM processing stage (details in § 4.1), which mitigates the error propagation to downstream devices.

3.2.3 Modeling Networks We construct the DeepQueueNet simulation network straightforwardly based on the target network topology. We achieve this by connecting pre-trained link models and device models, and form a DNN architecture that has an one-to-one correspondence to the target network at link-level, port-level, and device-level. We call the resultant DNN model a DeepQueueNet model. We run inference on this model to obtain predictions of packet latency for all incoming packets.

Algorithm 1: Iterative Re-Sequencing Algorithm (IRSA)

Input : G (Network Topology), I (Ingress Packet Stream)
Output: L (Packet Stream in Network)

```
/* Initial Inference                                   */
foreach device n in G do
    Perform n's model inference using I;
    Update n's egress packet stream in L;
end
/* Iterate until convergence                           */
for diameter (G) do
    Initiate L';
    foreach device n in G do
        Perform n's model inference using L;
        Update n's egress packet stream in L;
    end
end
def diameter(G):
    /* returns diameter of graph G.                    */
```

3.2.4 Running DeepQueueNet Processing packets in batches is the key factor to enable high scalability, but it also brings the "mis-batching" problem. To mitigate this problem, we design Iterative Re-Sequencing Algorithm (IRSA) as the core execution logic of DeepQueueNet.

We first examine the mis-batching problem. We split each port's ingress packet stream into batches based on a pre-defined time window, and we do not know the exact delays added to the upstream egress packets before the upstream device model finishes its inference. Therefore, we cannot directly concatenate the tensors trivially as the final results, because some packets may wrongly fall into the previous batch or the next batch, reducing the overall prediction accuracy. We name this the "mis-batching" problem.

We design IRSA to mitigate the impact of mis-batching. The key idea is to iteratively adjust the batches, and run the inference multiple times until convergence. As shown in Algorithm 1, at iteration t, all devices (including links) will pull the packet flows from iteration $t-1$ from their upstream devices. Once all the packet streams are collected, each device splits the ingress packet stream in batches, and performs an inference to obtain the output sequence of these packets. These output sequences will be pulled to their corresponding destination devices in the next iteration.

IRSA is the key execution logic of DeepQueueNet. It reorders the packets on each link and each device based on their time-stamps in the previous iteration, which gradually puts the packet in its correct batch on each link. Theorem 3.1 guarantees IRSA's convergence.

Theorem 3.1 (Convergence of IRSA). *Given a network topology G, and its diameter is d, IRSA is guaranteed to converge in d iterations.*

Proof of Theorem 3.1: Define $d_{node_i,iport_j}$=max{number of hops required to reach ingress port j of node i | paths passing ingress port j, node i} and $d_{node_i,eport_j}$=max{ $d_{node_i,iport_k}$+1 | ingress port k connected to egress port j, node i}. It's equivalent to prove that

(i) Flows to ingress port j, node i in a topology can be confirmed after $d_{node_i,iport_j}$ iterations;

(ii) Flows emitting from egress port j, node i in a topology can be confirmed after $d_{node_i,eport_j}$ iterations.

Since (ii) can be derived directly from (i), we would prove (i) only, which can be established by induction on the degree of ingress ports. If $d_{node_i,iport_j}$=0, i.e., ingress port j, node i is directly connected to servers, the proposition follows in this case. Assume that the proposition is valid for all ingress ports with degree $\leq n-1$ and consider the case of n. Notice that flows to ingress ports with degree of n are those emitting from egress ports with degree of n, which are directly connected to ingress ports with degree $\leq n-1$ and can be confirmed after $n-1$ times of iteration from the induction assumption. □

Theorem 3.1 provides an upper-bound on how many iterations are needed given a network topology, which limits the simulation completion time of DeepQueueNet. However, we also note that, in most cases, DeepQueueNet takes much fewer iterations to complete than the upper-bound.

4 ALGORITHMS & TECHNIQUES

In this section, we first describe the techniques and algorithms used in PTM: the pre-PTM feature engineering and data augmentation, the PTM DNN architecture, and the post-PTM error correction.

4.1 Pre-PTM Data Augmentation & Feature Engineering

We perform pre-processing of the ingress packet streams before using it in training, testing, and model inference of the PTM models. The first feature to add is the type of scheduler of the device, and we use one-hot encoding for this feature: 1000 for SP, 0100 for WRR, 0010 for DRR, and 0001 for WFQ. For SP schedulers, we add a priority to the packet vector, based on a flow ID to priority table. For WRR, WFQ, DRR schedulers, we add a weight to the packet vector, based on a flow ID to weight table. Both tables are assumed to be given before the simulation, and they can be described as the following equations:

$$priority(\text{fid}) \rightarrow \text{priority} \tag{8}$$

$$weight(\text{fid}) \rightarrow \text{weight} \tag{9}$$

We also add a workload feature to the packet vector, as a packet's sojourn time in a system depends on the state of the system when it arrives. The workload is an exponential moving average of bytes arriving at the ingress packet stream, and the smoothing factor is 0.95.

Finally, since the features extracted from the input data are of different scale, normalization is needed to scale the data between 0 and 1, which will also help in faster convergence. To normalize our data, we use the `MinMaxScaler` function provided by `scikit-learn`: $\frac{x_i - \min(x)}{\max(x) - \min(x)}$.

4.2 PTM Architecture

We model the packet streams as time series, and we use DNN as PTM to encapsulate the complex TM mechanisms of network devices, due to the exponential complexity of accurate queueing models. The PTM should predict the time series of sojourn times added to each of the event (packet) in the ingress time series, which is a sequence-to-sequence (seq2seq) processing task. For seq2seq tasks, various

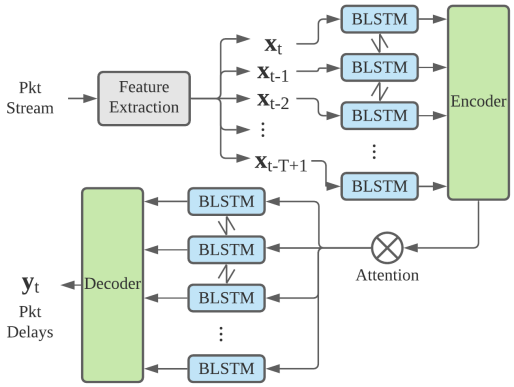


Figure 5: PTM DNN Architecture

neural network architecture have been proposed, such as recurrent neural network [44] and long-short term memory model [11, 18]. Among them, the Transformer architecture [9] with attention mechanism has demonstrated state-of-the-art performance in various seq2seq tasks [9]. We compared these approaches using traces of a single device, and in the end choose the Transformer architecture (Figure 5). To train the PTM, we follow the approach of regression in forecasting the delay a packet experienced in a device. We use the sojourn time of a packet in a device as the response variable and the features extracted from the feature extraction module as the predictors. We use ns.py[3] simulator to generate training data. ns.py is an open source network DES that we implemented using only Python. It is flexible and easy to use, especially for the study of queueing theory for single device or small topologies. The language choice of ns.py also makes it convenient to integrate with the deep learning and data analytic eco-system of Python. We have verified its simulation accuracy against ns3 [39] and an internal network processor simulator in Huawei which is cycle-accurate. Since we only need to conduct simulations with just one device to obtain the training data, we choose ns.py over "heavy-weight" C++-based simulators like ns3 or OMNet++ [46].

We perform extensive experiments to evaluate the PTM model in § 6, and it has shown high accuracy and generality. We believe this is because this architecture can capture relationships and correlations between packets, due to its multi-head attention mechanism and processing packet streams as a whole.

4.3 Post-PTM Statistical Error Correction

To control the propagation of error, we use a statistical error correction (SEC) method. Since the predicted sojourn time is added to the arrival time in the packet streams, the errors will propagate to the next devices, and accumulate along the path for all packets. Therefore, we must add a post-processing step for PTM to mitigate this effect. We design SEC based on observations of sojourn time prediction errors of different PTMs. Figure 6 shows three examples of relative error vs. predicted sojourn time for different schedulers. We make three observations: 1) the relative error of a single PTM model is not monotonic with respect to the predicted sojourn time; 2) for similar sojourn time predictions, their errors are also similar; 3) for a device model, the error distribution is stable for different schedulers and traffic generation patterns.

[3] https://github.com/TL-System/ns.py and https://pypi.org/project/ns.py/

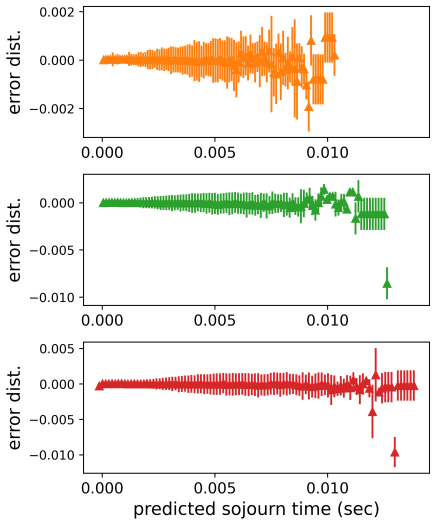


Figure 6: Statistical error distribution.

Thus we decide to 1) for each PTM, after its training has converged, we collect the errors for each sojourn time predictions (same as Figure 6); 2) we cluster the errors of nearby sojourn predictions into bins using DBSCAN algorithm [43], and 3) for sojourn time predictions falling into a bin, we subtract the average error of the bin from the prediction. SEC is based on the statistics of error distribution, which is a by-product of model training. We demonstrate the accuracy improvement of SEC in § 6.

5 PROTOTYPING DEEPQUEUENET

5.1 Implementation

We implement DeepQueueNet using Python 3.7 and TensorFlow 1.13.1. We follow the system architecture (Figure 2), and implement the techniques and algorithms in § 4.

The detailed workflow of our implementation is as follows: We implement training-related tooling and the model architecture in `DUtil`. We use `DUtil` to produce trained models, which are indexed and stored in `DLib` before we conduct a simulation. For each simulation, we expect the users to provide network topology, device configuration, and traffic generators as input. `SInit` processes the topology and device configuration, and then fetches the device models from `DLib`. It then composes the fetched models to form a DeepQueueNet model for this simulation. `TGUtil` is a factory for traffic generators, and can take packet capture (PCAP) files or MAP matrices as input for the generators. For example, if a PCAP file is provided, `TGUtil` produces a `TGen` generator, which parses the file to yield batched time series of packets arrivals. With the time series and the DeepQueueNet model of the target network, `SRun` performs inference of the model, and runs IRSA and SEC to refine the results. Finally, `SRun` outputs the packet-level time series for each link of the topology as simulation results.

5.2 Training DeepQueueNet

Due to resource limits, we conduct training and inference on two testbeds. We perform the training on our testbed with an Nvidia Tesla V100-32GB GPU (Testbed 1), and execute the inference phase

Time steps	21
BLSTM	(200,100)
Multi-head attention	3, (64,32)

Table 1: Hyper-parameters of DeepQueueNet.

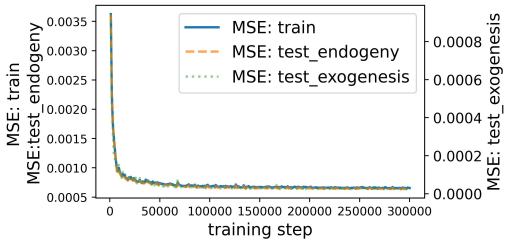


Figure 7: MSE over time for PTM Training.

on another testbed with 4 Nvidia Tesla V100-16GB GPUs (Testbed 2). We select a 2-layer BLSTM cell for the Decoder-Encoder components of the PTM model. We adopt 3 parallel heads to jointly attend to information from different representation subspaces at different positions. Table 1 shows the hyper-parameters of the model. We train PTM model on a collection with 3,500 packet streams sampled from the K-port switch in ns.py simulations. The packets arrival at the ingress ports are assumed to follow one of three stochastic processes: MAP, Poisson Process, and an On-Off process. We vary the intensity of these processes so that the load factor of each port of the device is in the range of [0.1,0.8]. Each packet stream lasts for $\max\{1.6e4, 4K \cdot 1e3\}$+Time steps−1 units of time, namely there are $\max\{1.6e4, 4K \cdot 1e3\}$ datapoints in each packet stream for sequence-to-sequence learning. Despite this dataset only contains samples from a K-port switch, it includes 3,500 randomly generated routing schemes and a wide variety of traffic matrices with different traffic intensity. We also configure the packet scheduler of the switch to enable FIFO, SP, DRR and WFQ disciplines. For SP, the packet's priority are randomly selected from 1 to 3. For DRR and WFQ, the weights are randomly selected from 1 to 9.

We train our model on 80% of the samples and evaluate it on the remaining 20%. For the exogenous evaluation, we randomly generate 8 extra packet streams with totally different configurations from the training set. During the training, we minimize the mean square error (MSE) between the predicted sojourn time and the ground truth. The loss function is minimized using the Adam optimizer with a fixed learning rate of 0.001. 256 samples from the training dataset is used to estimate the error gradient. In Testbed 1, this took around 1h40m for the device model of a 2-port switch to converge, 3h22m for a 4-port switch, and 11h24m for a 64-port switch. Figure 7 shows the loss during the training process of a 4-port switch. Table 2 shows the approximating precision of DeepQueueNet to a K-port switch with different packet schedulers. We measure the accuracy of the model using the normalised Wasserstein distance w_1=`W1(prediction,label)/W1([0]*len(label), label)`: if the predictions are accurate (close to ground-truth delays), then w_1 is close to 0 (the lower the better).

We observe that the training is stable and the loss drops quickly. The obtained PTM can achieve a DES-level accuracy in the simplest configuration. And the achieved accuracy drops with the increase of a device's complexity (K) due to the inherent uncertainty of the

	Device	No. of class	w_1	w_1 (Refined)
FIFO	2-port	1	0.006967	-
	4-port	1	0.022549	0.009185
	8-port	1	0.032825	0.016606
	16-port	1	0.047078	0.032306
	32-port	1	0.051662	0.049363
	64-port	1	0.053358	0.052345
Multi-level	4-port	2	0.028651	0.016296
	4-port	3	0.036128	0.024085

Table 2: Precision of DeepQueueNet to a K-port switch. The normalized Wasserstein distance w_1 is used to measure the accuracy. The final column in the table is the results of a K-port switch by simply doubling the time steps from 21 to 42.

model. We believe this kind of uncertainty can not be eliminated but can be mitigated by fine-tuning DeepQueueNet, due to the fact that we are processing packets in batches. For example, the sojourn time of a packet with lower priority depends on whether or not there are packets arriving at the system when it is in the system, which can be depicted by a random variable but by no means a definite value. We list the w_1 with refined data in Table 2, and we can see that the inference accuracy greatly improves, especially for $K \leq 16$.

6 EVALUATION

By construction, DeepQueueNet gains packet-level visibility: its DNN architecture maps directly to target network topology, and the simulation results are traces of packet streams, which can be analysed using arbitrary metric. Thus, in this section, we focus on verifying its accuracy, generality, and scalability. We summarize our results as follows:

Summary of Results:

- **Accuracy:** DeepQueueNet achieves superior accuracy for average and 99th percentile round-trip time (RTT) in all scenarios compared to state-of-the-art DNN-based EPEs [42].
- **Generalizability:** Using extensive experiment, we show that DeepQueueNet's estimation accuracy with respect to w_1 remains high across variations in topology, TM configurations, and traffic generation models, without need for model re-training.
- **Scalability:** We show that DeepQueueNet can be accelerated in parallel using multiple GPUs. We deploy DeepQueueNet on a 4-GPU cluster, and it demonstrates near-linear speed-up with the number of GPUs.

Setting. We perform experiments using trained DeepQueueNet models, and all experiments are done without model retraining. We use Testbed 2 described in § 5 to conduct the experiments, and we use the hyper-parameters. For all experiments, the ground-truth is generated using the ns.py simulator using the same settings. The links in the topology is 10Gbps.

Our main comparison targets are RouteNet [41, 42] for general topology and MimicNet [23, 50] for FatTree topology. For RouteNet, we follow the latest paper [42] to setup and train RouteNet. For MimicNet, we ensure that the traffic generation procedure is consistent with DeepQueueNet, and we run the default MimicNet model and training approach [50] to obtain the prediction model on an AWS p3.2xlarge instance (Tesla V100 GPU, 16GB, 8 vCPUs). We summarize the parameter setting for MimicNet in Table 3.

	`numToRsAndUplinks`	`numOfServersPerRack`	`numClusters`
FatTree16	2	4	2
FatTree64	4	4	4
FatTree128	4	4	8

Table 3: MimicNet's Parameter setting for different sizes of FatTree.

		avgRTT (w_1)	p99RTT (w_1)	avgJitter (w_1)	p99Jitter (w_1)
DQN	MAP	0.017	0.039	0.031	0.035
	Poisson	0.009	0.014	0.055	0.035
	Onoff	0.006	0.051	0.016	0.022
	BC-pAug89	0.003	0.012	0.009	0.007
	Anarchy	0.026	0.068	0.097	0.030
RN	MAP	0.044	0.014	0.041	0.030
	Poisson	0.674	0.972	1.951	1.083
	Onoff	0.549	0.578	1.420	0.847

Table 4: Generality for traffic generation models. DQN stands for DeepQueueNet; RN stands for RouteNet. The normalized Wasserstein distance w_1 is done path-wise.

6.1 Generality of DeepQueueNet

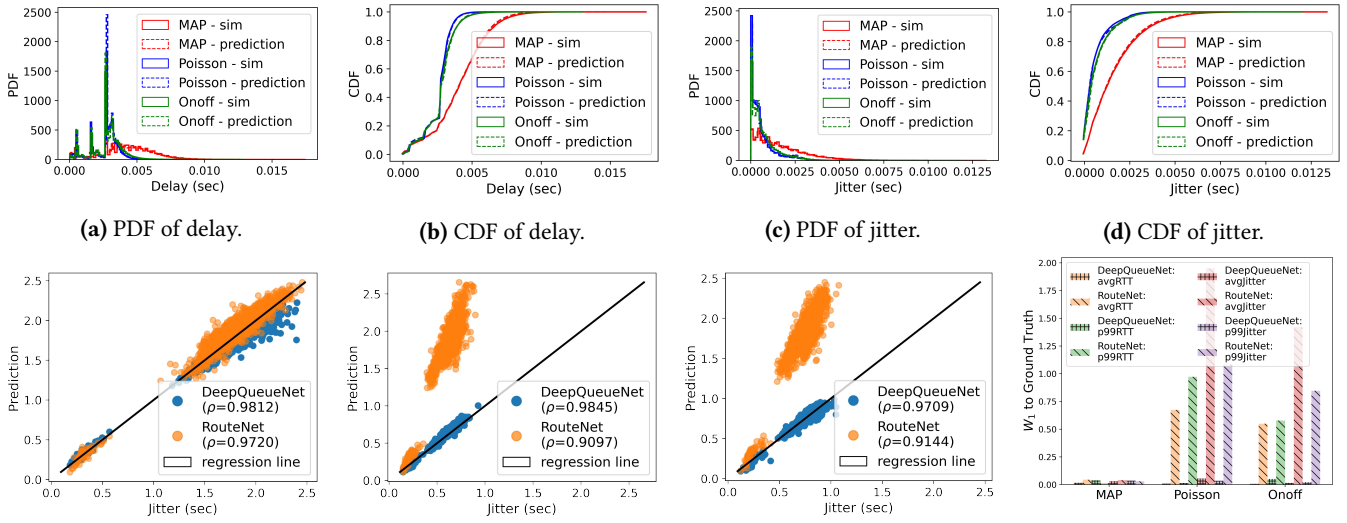
We evaluate DeepQueueNet's generality in three aspects: traffic generation models, topology, and TM mechanisms.

Generality for Traffic Generation Models. We use three traffic generation models: MAP (fit well with realistic traffic traces, described in Appendix A.1), Poisson (λ varies to generate 0.1 to 0.9 link load), and On-Off (with transition probability of 0.2 for On state and 0.5 for Off state). The sources and destinations of the traffic flows are selected uniformly at random.

Figure 8 shows the performance of DeepQueueNet in the baseline configuration - FIFO queuing discipline - for a FatTree(k=4) network with 16 servers (FatTree16). Also shown are results from RouteNet, which was trained on the FatTree16 network. Here, we explored three different traffic generation models with the same traffic matrix to evaluate how they affect the modeling capabilities of DeepQueueNet and RouteNet. We observe that DeepQueueNet achieves superior generality over RouteNet. RouteNet, due to its inherent drawback of using the traffic matrix as the input features, is virtually devoid of generality for traffic generation models. In Figure 8 (e,f,g), the x-axis is ground-truth latency and the y-axis is predicted latency. We can see that, if the prediction is accurate, the predicted latency should fall close to the solid line (y=x). In Figure 8 (f,g), when the traffic pattern is changed from MAP to Poisson and On-Off, RouteNet's predictions fall far-away from line (y=x), and DeepQueueNet's predictions fall close to it, showing higher accuracy and generality for traffic generation models.

We report the W_1 (Wasserstein distance) metric, which is a distance function defined between two distributions—in our case, the predicted latency distribution and the ground-truth. Table 4 is a summary of all the experiments we perform in Figure 8 together with the results of DeepQueueNet on traces from two public datasets: BC-pAug89 trace[4] and Anarchy trace[5]. Notably, the readout neural networks of RouteNet are connected to the path's and/or the link's hidden states. It can only estimate some path and/or link-level metrics. Since there exist multiple available paths for most of source-destination pairs in a network, it is almost impossible for RouteNet to provide an accurate quantile-based end-to-end

[4]The dataset is available at http://www.sfu.ca/ ljilja/TRAFFIC/Bellcore/.
[5]The dataset is available at https://datasets.simula.no/ao/.



(a) PDF of delay. **(b)** CDF of delay. **(c)** PDF of jitter. **(d)** CDF of jitter.

(e) MAP: avgJitter. **(f)** Poisson: avgJitter. **(g)** Onoff: avgJitter. **(h)** W_1: RouteNet vs. DeepQueueNet.

Figure 8: Generality for traffic generation models. The accuracy of DeepQueueNet in the baseline configuration (FIFO) for a FatTree16 network. Also shown are results from RouteNet. Since RouteNet can only output a certain path's delay/jitter (not end-to-end), all accuracy comparisons are done path-wise. The Pearson correlation ρ becomes unreliable when accessing RouteNet's generality for traffic generation models. Accuracy is then quantified via the Wasserstein distance (W_1) to the distribution observed in the original simulation (`W1(prediction,label)/W1([0]*len(label), label)`). Lower is better.

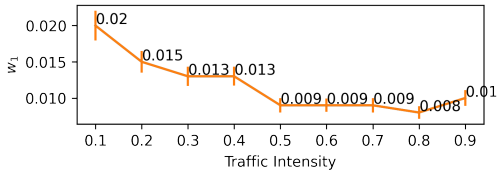


Figure 9: Inference accuracy of DeepQueueNet with different traffic intensity.

measure in a network, because quantile-based measures are not sub-additive.

Finally, we also conduct an experiment to show that DeepQueueNet can be generalized to dynamic load factors. Recall in the training data in § 5, the load factor of each port of the device is in the range of [0.1,0.8]. In these experiments, we increase the range of load factor variation to [0.1,0.9], and plot the inference accuracy in Figure 9. We observe that, for a previously unseen load factor of 0.9, DeepQueueNet can still achieve high accuracy (low normalized w_1 distance).

Topology Generality. We explore three different network topologies to show DeepQueueNet's generalizability: Line, 2dTorus, FatTree[6], as well as two Wide Area Networks from Internet Topology Zoo[7]: Abilene and GÉANT. These topologies present different connectivity in practical scenarios, which may affect the learning capability and estimation accuracy.

Tables 5 shows the summary of the obtained evaluation results in the baseline configuration coupled with a Poisson arrival process. In these results, we can observe that DeepQueueNet shows an obvious advantage over RouteNet, especially in Line graphs, which is the simplest cascaded network and can be simply treated as a path or a special case of some complex networks. Table 5 also shows the W_1 distance to the ground-truth distribution. We can see that the estimated distribution of DeepQueueNet is much closer to the ground-truth (W_1 is close to 0 for all cases), while RouteNet performs well only for one scenario. For varying scale of FatTree topology, we also compare against MimicNet. We observe that MimicNet achieves higher accuracy which is reasonable as it uses DES to accurately simulate a small subset of the network, and thus can obtain detailed information such as queueing and protocol interactions.

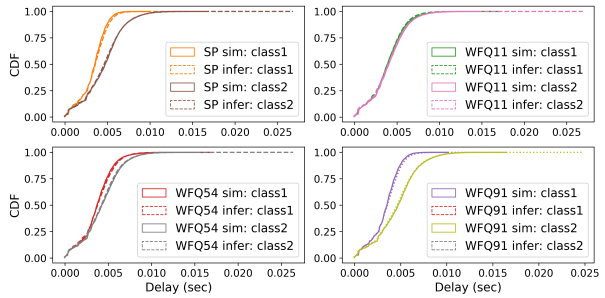
We credit the topology generalization capability of DeepQueueNet to SEC and IRSA. As an ablation study, we turn off SEC for Line6, FatTree64 and FatTree128, and the average RTT accuracy of DeepQueueNet degrades to 81%, 64%, and 13%, respectively. We also emphasize that IRSA cannot be turned off, because without IRSA, we cannot guarantee the correctness of simulation due to the mis-batching problem. With SEC and IRSA, a trained K-port switch PTM model can be used to construct any topology with node degree no more than K. As we have proven in§ 3.2.4, IRSA is guaranteed to converge after d iterations, where d is the diameter of the topology.

TM Generality. Unlike current EPEs, which are insensitive to the traffic management mechanisms, DeepQueueNet's PTM model is TM-aware due to the data augmentation procedures. To demonstrate generalizability for TMs, we use two type of packet schedulers, SP and WFQ (representing DRR and WRR). For two class WFQ, we vary the weight ratio between 1:1, 5:4, and 9:1. For 3-class WFQ, we use 1:1:1 as the weight ratio. For SP, we equally mark

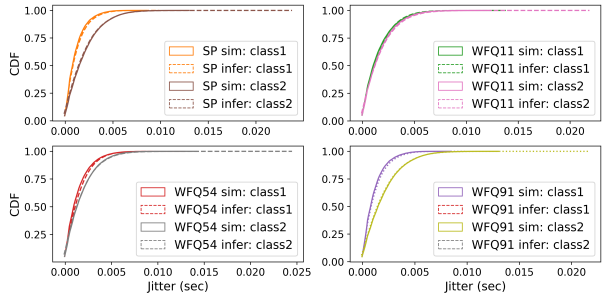
[6]FatTree16 is a FatTree(4) network with 16 servers; FatTree64 [30] is a FatTree (4-ary 3-tree) network with 64 servers; FatTree128 is a FatTree(8) network with 128 servers.
[7]http://www.topology-zoo.org/

		avgRTT (w_1)	p99RTT (w_1)	avgJitter (w_1)	p99Jitter (w_1)
DQN	Line4	0.0003	0.0023	0.0037	0.0000
	Line6	0.0002	0.0016	0.0041	0.0002
	Abilene	0.0017	0.0042	0.0137	0.0011
	GÉANT	0.0009	0.0013	0.0023	0.0032
	2dTorus(4x4)	0.0041	0.0274	0.0233	0.0158
	2dTorus(6x6)	0.0066	0.0277	0.0319	0.0156
	FatTree16	0.0086	0.0145	0.0548	0.0349
	FatTree64	0.0176	0.0395	0.0291	0.0215
	FatTree128	0.0133	0.0532	0.0395	0.0438
RN	Line4	0.5266	0.1978	5.3139	2.2943
	Line6	0.6561	0.2555	2.9424	1.4225
	Abilene	0.4781	0.1984	4.4904	5.5434
	GÉANT	0.4589	0.1328	4.1983	5.4555
	2dTorus(4x4)	1.2527	1.4644	5.5023	2.3050
	2dTorus(6x6)	1.1005	0.5959	4.6492	2.6154
	FatTree16	0.6737	0.9723	1.9510	1.0834
	FatTree64	0.1406	0.3370	5.2964	2.2873
	FatTree128	0.9824	0.6397	6.7684	1.8705
MN	FatTree16	0.0090	0.0135	0.1559	0.1172
	FatTree64	0.0167	0.0179	0.1687	0.0625
	FatTree128	0.0172	0.0194	0.1628	0.0667

Table 5: Topology Generality in the baseline configuration - FIFO + Poisson. DQN stands for DeepQueueNet; RN stands for RouteNet; MN stands for MimicNet. The normalized Wasserstein distance w_1 is done path-wise.



(a) CDFs of delays.



(b) CDFs of jitters.

Figure 10: The performance of DeepQueueNet on a FatTree16 network with different traffic management configurations.

the traffic flows with different priorities. We fix the topology to be FatTree16 and the traffic source for all flows is MAP.

We show the performance results in Figure 10 and Table 6. We observe that DeepQueueNet's predicted latency distribution is very close to the ground-truth, which shows that DeepQueueNet remains highly accurate for different configuration of packet schedulers.

6.2 Scalability of DeepQueueNet

DeepQueueNet has high scalability with respect to the size of simulation. This is because estimation using DeepQueueNet is simply

		avgRTT (w_1)	p99RTT (w_1)	avgJitter (w_1)	p99Jitter (w_1)
2-class	WFQ	0.020	0.046	0.037	0.042
	SP	0.023	0.050	0.040	0.045
3-class	WFQ	0.027	0.050	0.048	0.041
	SP	0.028	0.029	0.032	0.026

Table 6: TM Generality. The normalized Wasserstein distance w_1 is used to assess the performance of DeepQueueNet on End-to-End delay/jitter prediction.

topology	method	# GPUs	time	speedup
FatTree16	DES	0	2h22m11s	-
	MimicNet	1	4m2s	-
	DeepQueueNet	1	5m12s	baseline
	DeepQueueNet	2	2m45s	1.89-fold
	DeepQueueNet	4	1m27s	3.59-fold
FatTree64	DES	-	9h23m53s	-
	MimicNet	1	11m18s	-
	DeepQueueNet	1	29m17s	baseline
	DeepQueueNet	2	15m12s	1.93-fold
	DeepQueueNet	4	8m5s	3.62-fold
FatTree128	DES	0	20h15m39s	-
	MimicNet	1	11m34s	-
	DeepQueueNet	1	1h6m18s	baseline
	DeepQueueNet	2	33m23s	1.99-fold
	DeepQueueNet	4	17m5s	3.88-fold

Table 7: Inference execution time with parallelization for 30s of simulated time on different FatTree networks (288,192 packets for a FatTree16 network, 1,152,768 packets for a FatTree64 network, and 2,305,536 packets for a FatTree128 network).

model inference, and we can run model inference in parallel on multiple devices or machines using distributed runtime of popular machine learning frameworks. In this way, we circumvent the key obstacle for DES, and can be accelerated in parallel. We showcase parallel acceleration of FatTree16/128 topologies on DeepQueueNet in Table 7. For each network configuration, we run DeepQueueNet, OMNet++ [46], and MimicNet over the same sets of generated workloads. We no longer use ns.py in this experiment because it is less performant compared to OMNet++ due to the language choice of Python. We evenly divide the DeepQueueNet model to the GPUs, and the division for FatTree16 topology on 4 GPUs is shown in Figure 11 as an example. Currently, we perform the division by hand, and we leave the automatic partitioning and parallelization of DeepQueueNet as future work.

In all systems, simulation time consists of both setup time, which is used to construct the network, allocate resources, and schedule the traffic, as well as the actual simulation. DeepQueueNet substantially speeds up both phases compared to DES. The running time of OMNet++ (the DES rows in Table 7) is shown for reference only, and it is not a fair comparison because of the computational power differences between CPU and GPU. The running time of MimicNet outperforms DeepQueueNet for all scales of FatTree topology using only a single GPU, and this is expected because MimicNet is optimized for this topology and DeepQueueNet needs to perform iterative procedures (IRSA) to guarantee correctness. Looking only at DeepQueueNet, our results show that DeepQueueNet can provide near-linear speed-ups with the number of additional GPUs. This is because we are only running inference on the GPUs, and the time complexity is almost constant.

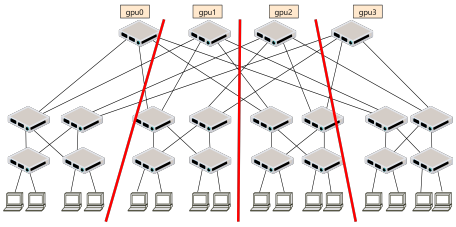


Figure 11: Example: Network decomposition for model-parallel inference of DeepQueueNet on 4 GPUs (FatTree16).

7 DISCUSSION

We have shown that DeepQueueNet helps network operators and designers answer the following question: given a packet trace, a network topology, and the TM configuration of the devices in the network, what will the output packet trace be for each link in the topology? In this section, we discuss DeepQueueNet's limitations and future directions.

Modeling State-ful Behaviors. DeepQueueNet is unable to deal with state-ful behaviors of network devices. Because DeepQueueNet only looks at the network layer, and indeed, ignores the complex interactions among the higher layers, such as the transport protocols they uses, the network layer TM mechanisms, as well as any possible interaction between devices. We make this intentional trade-off favoring scalability over accuracy, and the same trade-off is also made by prior works that uses DNN to accelerate performance estimation [23, 26, 42, 50]. Inspired by the flow-level feeder models in MimicNet [50], one promising future direction is to model the state-ful behavior of network participants using DNN-based traffic generation models, and integrate them with DeepQueueNet to compose an comprehensive simulator. This is made possible because DeepQueueNet does not put any limit on the traffic generation model and only processes packet traces.

Usage Scenarios. DeepQueueNet focuses on modeling the performance distribution of a network when it reaches stability, and in general ignores the transient states of the network. This accelerates the simulation of the network. We believe that DeepQueueNet is useful for scenarios where operators care more about the behavior of the network at stability, such as capacity planning and topology design. In addition, since a DeepQueueNet model is fully differentiable, we can combine it with a gradient-descent-based algorithm to optimize parameters of network devices [28], and we leave this as future work.

Accommodating New Devices. DeepQueueNet is able to perform single-device black-box modeling of any new TM mechanisms, including queueing disciplines. We are able to train a black-box model for any device given its input packet trace and output packet trace. The device model can then be used to form DeepQueueNet models of an entire network, so that the designer of the new mechanism can inspect its performance at a larger scale. In our experience with network operators, they are usually more interested in performance of a larger network than that of a single device. However, this model is limited. For example, any network device/middle-box that combines or splits packets, are hard to model with the device model of DeepQueueNet, and we leave modeling of general packet-processing device as future work.

8 RELATED WORKS

We identify two research areas are closely related to DeepQueueNet: network simulation and DNN explainability.

As discussed in § 2, network simulation solutions have three types: DES, continuous simulators, and DNN-based EPEs. The main problem of DES [21, 29, 39, 46] is scalability. DeepQueueNet overcomes this issue by incorporating continuous simulation techniques, and use DNN to approximate latency functions which can be deployed to parallel computing clusters. For continuous simulation, both control-theoretic simulators [2, 31, 34, 37] and network calculus cannot provide useful statistics for network engineering, and the computational cost of queue-theoretic performance estimator is too high. DeepQueueNet carefully identify the mathematically-intractable and computationally-expensive parts of queueing-theoretic models, and model them using DNNs, which has constant time complexity during inference. Finally, existing EPEs [41, 42] are neither visible at packet-level nor generalizable, as we have shown in our evaluations. DeepQueueNet, by design, has packet-level visibility. Recently, to enhance visibility and generality of EPEs, MimicNet [23, 50] propose to limit the application of DNN from network-scale to cluster-scale. However, MimicNet only works for FatTree [30], and focuses on scale generalizability. As our experiments have shown, DeepQueueNet can generalize to arbitrary topology, popular TM mechanisms, and different traffic generation models.

Visibility is closely linked to model explainability of DNN [10, 12, 47], and the design of DeepQueueNet is also inspired by differentiable programming [17, 19], especially its usage in physics simulations [27]. We do not claim contribution to this area. Instead, our work is an application of the principles of explainable artificial intelligence, and we focus on packet-level visibility in the design of DeepQueueNet. Structurally, DeepQueueNet model has one-to-one correspondence with the target network topology; the output traces of each device is simply a packet trace, on which any new metric can be applied without retraining the model.

9 CONCLUDING REMARKS

We present DeepQueueNet, a scalable and generalizable network performance estimator with packet-level visibility by combining scalable DNN-based continuous simulation with discrete event simulation. In contrast to prior solutions, we limit the use of DNN to the modeling of device-local TM mechanisms, guided by a solid queueing-theoretic modelling of modern packet schedulers. In this way, we directly map target network topology to form DeepQueueNet models. Experiments show that our prototype can achieve near-linear speed-up, while maintaining high accuracy across all scenarios.

Acknowledgements: We thank our shepherd Prof. Vincent Liu and the anonymous SIGCOMM reviewers for their constructive feedback and suggestions. We also thank Dr. Qizhen Zhang for his help with the evaluation. This work is supported in part by funding from the Huawei-CUHK Joint PhD Program (6906388), Jinan Scientific Research Leader Studio Project (2021GXRC091), and National Natural Science Foundation of China (61802233).

REFERENCES

[1] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 1–1.

[2] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review* 39, 1 (2011), 73–84.

[3] Søren Asmussen and Ger Koole. 1993. Marked point processes as limits of Markovian arrival streams. *Journal of Applied Probability* (1993), 365–372.

[4] F. Baccelli and D. Hong. 2003. Flow level simulation of large IP networks. In *Proc. IEEE INFOCOM*, Vol. 3. 1911–1921.

[5] Giuliano Casale, Eddy Z Zhang, and Evgenia Smirni. 2008. KPC-toolbox: Simple yet effective trace fitting using Markovian arrival processes. In *Proc. IEEE Int. Conf. Quantitative Evaluation of Systems*. St. Malo, France, 83–92.

[6] Srinivas R Chakravarthy. 2010. Markovian arrival processes. *Wiley Encyclopedia of Operations Research and Management Science* (2010).

[7] Srinivas R Chakravarthy, Shruti, and Alexander Rumyantsev. 2020. Analysis of a Queueing Model with Batch Markovian Arrival Process and General Distribution for Group Clearance. *Methodology and Computing in Applied Probability* (2020), 1–29.

[8] Florin Ciucu and Jens Schmitt. 2012. Perspectives on network calculus: no free lunch, but still good value. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 311–322.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Krishna Gade, Sahin Cem Geyik, Krishnaram Kenthapadi, Varun Mithal, and Ankur Taly. 2019. Explainable AI in industry. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3203–3204.

[11] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[12] David Gunning. 2017. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web* 2, 2 (2017).

[13] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. 2015. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[14] Qi-Ming He. 2014. *Fundamentals of matrix-analytic methods*. Vol. 365. Springer, New York.

[15] Gábor Horváth, B. Van Houdt, and M. Telek. 2014. Commuting Matrices in the Queue Length and Sojourn Time Analysis of MAP/MAP/1 Queues. *Stochastic Models* 30, 4 (2014), 554–575.

[16] Gábor Horváth and Hiroyuki Okamura. 2013. A fast EM algorithm for fitting marked Markovian arrival processes with a new special structure. In *Proc. Eur. Workshop Perform. Engineering*. Springer, Berlin, Heidelberg, Venice, Italy, 119–133.

[17] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935* (2019).

[18] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991* (2015).

[19] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. 2019. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587* (2019).

[20] Teerawat Issariyakul and Ekram Hossain. 2009. Introduction to network simulator 2 (NS2). In *Introduction to network simulator NS2*. Springer, 1–18.

[21] Teerawat Issariyakul and Ekram Hossain. 2009. Introduction to network simulator 2 (NS2). In *Introduction to network simulator NS2*. Springer, Boston, MA, 1–18.

[22] Shafagh Jafer, Qi Liu, and Gabriel Wainer. 2013. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory* 30 (2013), 54–73.

[23] Charles W. Kazer, João Sedoc, Kelvin K.W. Ng, Vincent Liu, and Lyle H. Ungar. 2018. Fast Network Simulation Through Approximation or: How Blind Men Can Describe Elephants. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 141–147.

[24] Alexander Klemm, Christoph Lindemann, and Marco Lohmann. 2003. Modeling IP traffic using the batch Markovian arrival process. *Performance Evaluation* 54, 2 (2003), 149–173.

[25] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network calculus: a theory of deterministic queuing systems for the internet*. Vol. 2050. Springer Science & Business Media.

[26] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. 2022. SimNet: Accurate and High-Performance Computer Architecture Simulation using Deep Learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–24.

[27] Hai-Jun Liao, Jin-Guo Liu, Lei Wang, and Tao Xiang. 2019. Differentiable programming tensor networks. *Physical Review X* 9, 3 (2019), 031041.

[28] Libin Liu, Li Chen, Hong Xu, and Hua Shao. 2020. Automated Traffic Engineering in SDWAN: Beyond Reinforcement Learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 430–435.

[29] Zheng Lu and Hongji Yang. 2012. *Unlocking the Power of OPNET Modeler*. Cambridge University Press, New York, NY.

[30] M.Alonso, S.Coll, J.M.Martínez, V.Santonja, and P.López. 2015. Power consumption management in fat-tree interconnection networks. *Parallel Comput.* 48 (2015), 59–80.

[31] Marco Ajmone Marsan, Michele Garetto, Paolo Giaccone, Emilio Leonardi, Enrico Schiattarella, and Alessandro Tarello. 2005. Using partial differential equations to model TCP mice and elephants in large IP networks. *IEEE/ACM Transactions on Networking* 13, 6 (2005), 1289–1301.

[32] Hiroyuki Masuyama and Tetsuya Takine. 2003. Sojourn time distribution in a MAP/M/1 processor-sharing queue. *Operations Research Letters* 31, 5 (2003), 406–412.

[33] S McCanne. 2006. Libpcap. *ftp://ftp. ee. lbl. gov/libpcap. tar. Z.* (2006).

[34] Vishal Misra, Wei-Bo Gong, and Don Towsley. 2000. Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 151–160.

[35] Jason C Neumann. 2015. *The book of GNS3: build virtual network labs using Cisco, Juniper, and more*. No Starch Press.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[37] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H Low. 2014. Multipath TCP: Analysis, design, and implementation. *IEEE/ACM Transactions on networking* 24, 1 (2014), 596–609.

[38] Xi Peng, Fan Zhang, Li Chen, and Gong Zhang. 2021. A MAP-based Performance Analysis on 5G-powered Cloud VR Streaming. In *Proc. IEEE International Conference on Communications (ICC)*. 1–6.

[39] George F Riley and Thomas R Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*. Vol. 14. Springer, Berlin, Heidelberg, 527.

[40] Thomas G Robertazzi. 2012. *Computer networks and systems: queueing theory and performance evaluation*. Springer Science & Business Media, New York, NY.

[41] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2019. Unveiling the Potential of Graph Neural Networks for Network Modeling and Optimization in SDN. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR'19)*. Association for Computing Machinery, New York, NY, 140–151.

[42] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio. 2020. RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2260–2270.

[43] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[44] Alex Sherstinsky. 2020. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena* 404 (2020), 132306.

[45] Han Vanholder. 2016. Efficient inference with tensorrt. In *GPU Technology Conference*, Vol. 1. 2.

[46] Andras Varga. 2019. A practical introduction to the OMNeT++ simulation framework. In *Recent Advances in Network Simulation*. Springer, Switzerland, 3–51.

[47] Xiang Wang, Xiangnan He, Fuli Feng, Liqiang Nie, and Tat-Seng Chua. 2018. Tem: Tree-enhanced embedding model for explainable recommendation. In *Proceedings of the 2018 World Wide Web Conference*. 1543–1552.

[48] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. 2021. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence* 2, 2 (2021), 109–127.

[49] Hyeon-Joong Yoo. 2015. Deep convolution neural networks in computer vision: a review. *IEIE Transactions on Smart Processing and Computing* 4, 1 (2015), 35–43.

[50] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 287–304.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A MAP-BASED MODELING FOR NETWORK TRAFFIC

Queueing modeling lays the foundation of theoretically analyzing the network performance, and is independent of the implementation of a DES. Since the origin of queueing theory was activated by A.K. Erlang in 1909, myriad models have been proposed for understanding and analyzing various networks, such as M/M/1 and GI/G/1 for single queueing systems and Jackson networks for the network of queues. Although general queueing models like GI/G/1 have few conditions, they are not tractable and thus difficult to obtain analytical or numerical results on performance measures of interest, such as the latency distribution and queue-length distribution. Classical tractable queueing models extensively uses Poisson processes to model the arrival process and the service process, such as the M/M/1 and Jackson networks, but such models turn out to be too simplistic in real networks. The Markovian arrival process (MAP) and its extensions are regarded as the most expressive models that are analytically tractable by using matrix-analytic methods. They have attracted much attention in the quantification of network performance [14, 15, 24, 32, 38]. When we sort the family of MAPs in ascending order of complexity, the list covers the Poisson process, the phase-type (PH) renewal process, the Markov modulated Poisson process (MMPP), the MAP and the batch MAP (BMAP). The same is true of representational capacity. Here we take the MAP as an example to show its application to real networks.

A.1 Model of inter-arrival times

The MAP can model the network traffic at a fine granularity by describing inter-arrival times (IATs) between packets. The MAP is governed by an underlying continuous-time Markov chain (CTMC) with finite states. Let $\mathbb{J}=\{1,2,...,M\}, M\geq 1$, be the state space of this CTMC. Let $\boldsymbol{D}_0$ and $\boldsymbol{D}_1$, $M\times M$ matrices, denote rate matrices of state transitions with zero and one arrival, respectively. Matrix $\boldsymbol{D}_0$ is non-singular and its diagonal entries are negative. The other entries of $\boldsymbol{D}_0$ and all entries of $\boldsymbol{D}_1$ are non-negative. The infinitesimal generator of the CTMC is given by $\boldsymbol{D}_0+\boldsymbol{D}_1$, and its row sum is zero by definition and hence a MAP has $(2M^2-M)$ free parameters.

A critical problem of applying a MAP-based model is to effectively estimate parameters fitted to the observed data. There are two categories of methods: the moment-matching (MM) method [5] and the maximum likelihood estimation (MLE) [16]. The MLE usually gives a better fitting performance at the expense of higher computational complexity. The idea of MLE is to estimate parameters by maximizing a likelihood function of the probability distribution of IATs, so that the observed data is the most probable under the assumed MAP model. Applying the expectation-maximization (EM) algorithm, this problem can be efficiently solved. Moreover, if $\boldsymbol{D}_0$ and $\boldsymbol{D}_1$ are assumed to have special structures, the solution will be more efficient [16].

With $\boldsymbol{D}_0$ and $\boldsymbol{D}_1$, the cumulative distribution function (CDF) of IATs [32] is explicitly give by

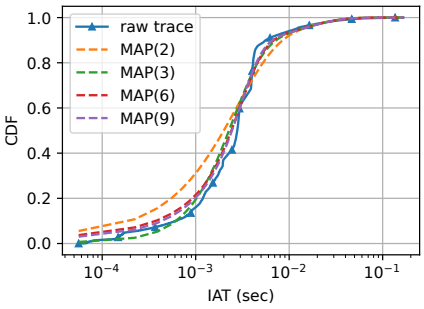
$$F(t)=1-\boldsymbol{\pi}_a e^{\boldsymbol{D}_0 t}\mathbf{1}, \quad t\geq 0,$$

where $\boldsymbol{\pi}_a$ is the unique solution of the linear system

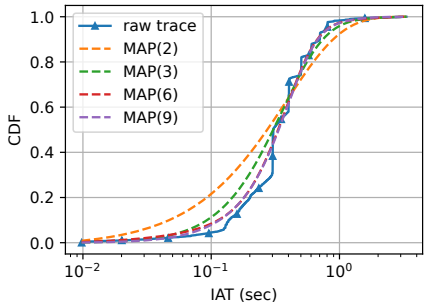
$$\begin{cases} \boldsymbol{\pi}_a(-\boldsymbol{D}_0)^{-1}\boldsymbol{D}_1=\boldsymbol{\pi}_a \\ \boldsymbol{\pi}_a\boldsymbol{e}=1 \end{cases}.$$

We show the effectiveness of MAP modeling by fitting real-world traffic traces from two public datasets: BC-pAug89 and Anarchy, respectively. Figure 12 depicts the fitting performance of MAP models by comparing the CDF of IATs. We observe that MAP models well capture the characteristics of the raw traffic. A higher dimensional MAP model improves the fitting accuracy, but suffers from higher computational complexity and overfitting. Our experiment shows that a moderate dimension could achieve a satisfactory performance.

In a nutshell, the MAP modeling is a powerful tool for describing complex traffic. For training DeepQueueNet, we utilize MAP models to generate additional training datasets.



(a) BC-pAug89 Trace



(b) Anarchy Trace

Figure 12: Fitting real traces with MAP models.

A.2 Model of service times

The service times t_s of a packet is determined by $t_s=\frac{L}{\mu}$, where L the packet size and μ is the service rate. The distribution of service times exactly reflects the distribution of packet sizes if the network uses constant service rate. The family of MAPs can also be used to describe service times. In practice, since service times is not as sophisticated as IATs, the most common type of service times is exponential distribution, which means the service occurs according to a Poission process. Other types like PH process and Markovian service process (MSP) have also been employed [15]. There are also

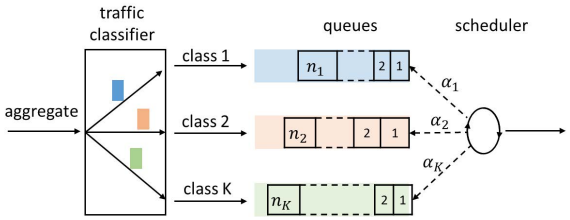


Figure 13: A multi-queue packet scheduler.

studies on general distributed service times, but they are usually intractable [7].

B STATE-AWARE MODELING OF NETWORK PACKET SCHEDULERS: SP/DRR/WFQ

A packet scheduler plays the role of an arbiter in packet switching network, which provides specific reordering of the queueing packets by using scheduling disciplines such as SP, DRR, WFQ, etc. In prior works, the analysis of packet schedulers are based on the PGPS where the arrivals are assumed to form a Poisson process and all buffers are always non-empty. These assumptions are too strong for real scenarios where the packet arrivals are bursty and correlated, and the scheduler skips empty queues and only serves non-empty ones in every round. Here we propose a fine-grained queueing model of multi-queue packet scheduler. Our work extends the representational capacity and enhances the accuracy of classical PGPS modeling, while remains tractable.

B.1 Queueing-theoretic modeling

Figure 13 illustrates how multi-class flows share the available link bandwidth (i.e., service rate) of a multi-queue packet scheduler. The incoming aggregate flow consists of K classes and is classified into K separate flows according to their classes. Each class maintains a separate queue, which is assumed to have infinite buffer space. The discipline of each queue is FIFO. In each round, the head-of-line packets of non-empty queues will be selected to transmit based on a specific scheduling discipline, for example, the WFQ with weights $[\alpha_1,\alpha_2,...,\alpha_K]$.

B.1.1 Model of multi-class arrivals We characterize the aggregate network flow by using the MAP with the infinitesimal generator $\boldsymbol{D}=\boldsymbol{D}_0+\boldsymbol{D}_1$, and the state space $\mathbb{J}=\{1,...,M\}$, where $\boldsymbol{D}_0=\left[d_{jk}^0\right]_{j,k=1}^M$ and $\boldsymbol{D}_1=\left[d_{jk}^1\right]_{j,k=1}^M$ are $M\times M$ matrices. The probability of the k-th class is $p_k>0$, which stands for the arriving rate proportion of each class in the aggregate flow. It holds that $\sum_{k=1}^K p_k=1$. Therefore, the arrivals of the k-th class are characterized by

$$\begin{cases}\boldsymbol{D}_0^{[k]}=\boldsymbol{D}_0+(1-p_k)\boldsymbol{D}_1\\ \boldsymbol{D}_1^{[k]}=p_k\boldsymbol{D}_1.\end{cases}$$

Let row vector $\boldsymbol{\pi}$ denote the stationary probability vector of the underlying CTMC, which is uniquely determined by $\boldsymbol{\pi}(\boldsymbol{D}_0+\boldsymbol{D}_1)=\boldsymbol{0}$ and $\boldsymbol{\pi e}=1$. The mean packet arrival rate of aggregate flow and of the k-th class are given by $\lambda=\boldsymbol{\pi}\boldsymbol{D}_1\boldsymbol{e}$ and $\lambda_k=p_k\lambda$, respectively.

B.1.2 Model of state-aware scheduling We first look into the WFQ, WRR and DRR disciplines. For tractability, we consider service times to be exponentially distributed. Service rate μ is assigned to all classes in proportion to corresponding weights $[\alpha_1,\alpha_2,...,\alpha_K]$, which can be regarded as the weights used in WFQ and WRR or the quanta used in DRR. When some queues are empty, the service rate will only be allocated among the nonempty queues. In practice, these scheduling disciplines achieve almost the same allocation effect in the long run. In the sequel, we refer all these disciplines as the WFQ to simply the exposition. Then we consider the SP discipline where the lower priority is starved until all the higher priority queues are empty. In both WFQ and SP, the actual scheduling depends on queue states, which motivates us to develop a state-aware scheduling model.

Let us denote the queue dynamics as a row vector

$$\boldsymbol{n}=\left[n_1,n_2,...,n_K\right],$$

where $n_k\in\mathbb{N}_0$ is the queue length of class k. The actual service rates allocated to class k, denoted by g_k, is a function of $\boldsymbol{n}$. For WFQ, we have

$$g_k(\boldsymbol{n})=\frac{\alpha_k\mathbf{1}_{\{n_k>0\}}}{\sum_{i=0}^K\alpha_i\mathbf{1}_{\{n_i>0\}}}\mu,$$

and for SP, we have

$$g_k(\boldsymbol{n})=\mu\mathbf{1}_{\{n_k>0 \text{ and } n_i=0,\forall i<k\}},$$

where $\mathbf{1}_{\{\cdot\}}$ is the indicator function. Besides the solution of WFQ and SP, our method can also be extended to other schedulers.

B.2 LDQBD reformulation

With MAP-based arrivals and the state-aware scheduling model, the queueing system is governed by the CTMC $Z(t)=\{\boldsymbol{n}(t),j(t),t\geq 0\}$, and the state space is given by $\mathbb{N}_0^K\times\mathbb{J}$, where $\times$ is the Cartesian product operator. We enumerate all states based on a well-designed order. The summation of all queue-lengths is defined as the level of this system, that is, $l=\boldsymbol{ne}$. We utilize a level-ascending-state-descending sequencing for all permutations of $\boldsymbol{n}$, and an ascending sequencing for the MAP states j. Take the case of two-class scheduling as an example. Its states can be listed as

$$\begin{aligned}&l=0:(0,0,1),...,(0,0,M);\\ &l=1:(1,0,1),...,(1,0,M);(0,1,1),...,(0,1,M);\\ &l=2:(2,0,1),...,(2,0,M);(1,1,1),...,(1,1,M);\\ &\quad(0,2,1),...,(0,2,M);\\ &\quad...\end{aligned}$$

The first two digits stand for queue-length states n_1 and n_2, and the last digit represents the MAP state j.

In general, the number of possible permutations of queue-length states is given by the binomial coefficient

$$c_l=\binom{l+K-1}{K-1}=\frac{(l+K-1)!}{l!(K-1)!}.$$

Since the size of the MAP state space is M, the number of all states at level l is obtained as $d_l=Mc_l$.

In order to achieve a tractable analysis, we reformulate the CTMC into a level-dependent quasi-birth–death (LDQBD) process, where the transitions from level l are categorized into three kinds:

(1) to states in level l;
(2) to states in level $(l-1)$;
(3) to states in level $(l+1)$.

Therefore, $Z(t)$ has a block-tridiagonally structured generator:

$$Q=\begin{bmatrix} Q_{00} & Q_{01} & O & O & O & O & O & \cdots \\ Q_{10} & Q_{11} & Q_{12} & O & O & O & O & \cdots \\ O & Q_{21} & Q_{22} & Q_{23} & O & O & O & \cdots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \cdots \\ O & O & O & O & Q_{l,l-1} & Q_{l,l} & Q_{l,l+1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots \end{bmatrix}$$

where $Q_{l,l+1}\in\mathbb{R}^{d_l\times d_{l+1}}, l\geq 0$, is a set of transitions form level l to level $(l+1)$, $Q_{l,l}\in\mathbb{R}^{d_l\times d_l}, l\geq 0$, is a set of transitions within level l, and $Q_{l,l-1}\in\mathbb{R}^{d_l\times d_{l-1}}, l\geq 1$, is a set of transitions form level l to level $(l-1)$. The ordering of states with a level is fixed up to a permutation. Note that $Q_{l,l-1}$ and $Q_{l,l+1}$ are non-negative rectangular matrices, and $Q_{l,l}$ is non-singular square matrix with negative diagonal entries and non-negative off-diagonal entries. Level 0 is the boundary level and has two non-zero blocks. The state transition rates of the LDQBD are given by:

(1) In $Q_{l,l+1}, l\geq 0$: $(\boldsymbol{n},j)\rightarrow(\boldsymbol{n}+\boldsymbol{e}_i,k)$ at $p_i d^1_{jk}$;
(2) In $Q_{l,l-1}, l\geq 1$: $(\boldsymbol{n},j)\rightarrow(\boldsymbol{n}-\boldsymbol{e}_i,j)$ at $g_i(\boldsymbol{n})$, if $\boldsymbol{n}-\boldsymbol{e}_i\in\mathbb{N}_0^K$;
(3) In $Q_{l,l}, l\geq 0$: $(\boldsymbol{n},j)\rightarrow(\boldsymbol{n},k)$ at d^0_{jk}, if $j\neq k$; and $(\boldsymbol{n},j)\rightarrow(\boldsymbol{n},j)$ at $d^0_{jj}-\sum_{k=1}^K g_k(\boldsymbol{n})$;
(4) All the remaining entries of Q are zeros.

Note that the row sum of $Q_{l,l-1}+Q_{l,l}+Q_{l,l+1}$ is zero.

Suppose the queueing system is stable, that is, the utilization $\rho=\frac{\lambda}{\mu}\leq 1$. The stationary probability vector of the LDQBD process is denoted by a row vector

$$\boldsymbol{\phi}=[\boldsymbol{\phi}_0,\boldsymbol{\phi}_1,\boldsymbol{\phi}_2,\ldots],$$

where $\boldsymbol{\phi}_l\in\mathbb{R}^{d_l}$ is the stationary probability vector of level l, and we have $\boldsymbol{\phi}_l=\{[\phi_{\boldsymbol{n},1},\ldots,\phi_{\boldsymbol{n},M}],\boldsymbol{n}\in\mathbb{N}_0^K,|\boldsymbol{n}|=l\}$. The stationary distribution vector $\boldsymbol{\phi}$ is uniquely determined by the linear system

$$\begin{cases}\boldsymbol{\phi}Q=\boldsymbol{0}\\ \boldsymbol{\phi}\boldsymbol{e}=1\end{cases}$$

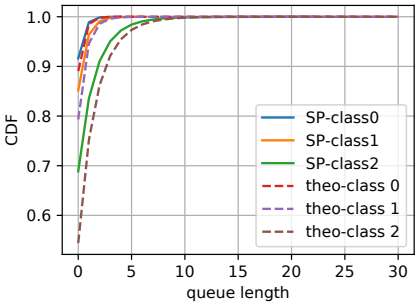
Based on $\boldsymbol{\phi}$, we can compute the marginal distribution of queue-length for each class. Here we skip the details of our algorithm to solve the linear system, since it is beyond this paper. The main idea is to apply the matrix continued fractions (MCFs) and to design an numerical iterative algorithm. The computational complexity is $O\left(M^3L^{3K}\right)$, and therefore is not feasible for solving large-scale scheduling system.

B.3 Numerical examples

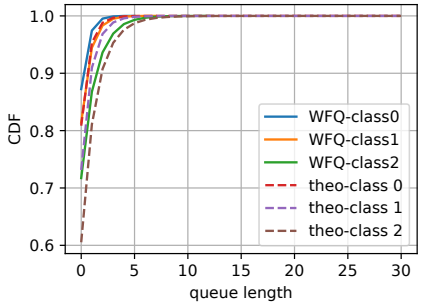
Here we validate our modeling and solution by comparing our result with the DES result. In this example, there are three classes in an aggregate flow and their proportion is 20%, 30% and 50%. The MAP(2) representation of the aggregate flow's IATs is given by

$$D_0=\begin{bmatrix}-12000 & 0\\ 0 & -3000\end{bmatrix} \quad \text{and} \quad D_1=\begin{bmatrix}3600 & 8400\\ 2100 & 900\end{bmatrix}.$$

The average arriving rate of the aggregate flow is 4800 packets per sec according to the MAP(2) model. The packet size is assumed to be constant 1426 bytes and the service rate to be exponentially distributed with mean rate of 100 Mbps. Figure 14 shows the queue-length CDFs under SP and WFQ (1:1:1). The solid lines stand for the empirical CDFs given by DES, while the dash lines are the queueing-theoretic CDFs.



(a) SP



(b) WFQ

Figure 14: Queueing performance of schedulers.

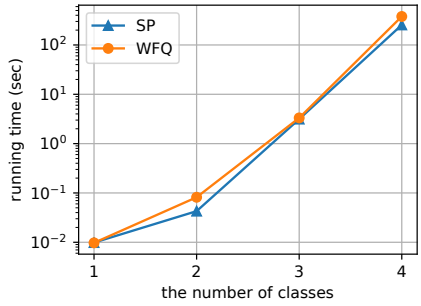


Figure 15: Running time of different scheduling cases.

As reflected in Figure 15, the running time increases exponentially as the number of classes rises, which coincides our complexity analysis. It demonstrates that the scalability of the tractable queueing model is a major challenge, which hinders its application to real networks of thousands or even millions of queues. In conclusion, queueing models give a theoretical insight into the system dynamics and an accurate estimation. But the exact analysis of queueing networks under real traffic becomes quickly intractable due to the state explosion.

		avgRTT		p99RTT		avgJitter		p99Jitter	
		ρ	95% CI	ρ	95% CI	ρ	95% CI	ρ	95% CI
DeepQueueNet	MAP	0.994	[0.992,0.995]	0.979	[0.974,0.983]	0.981	[0.975,0.985]	0.966	[0.958,0.973]
	Poisson	0.999	[0.999,0.999]	0.988	[0.985,0.990]	0.985	[0.981,0.987]	0.978	[0.973,0.982]
	Onoff	0.997	[0.996,0.998]	0.950	[0.931,0.961]	0.971	[0.963,0.976]	0.932	[0.915,0.945]
	BC-pAug89	0.999	[0.999,0.999]	0.983	[0.979,0.986]	0.981	[0.975,0.985]	0.972	[0.965,0.978]
	Anarchy	0.997	[0.996,0.997]	0.965	[0.956,0.972]	0.937	[0.922,0.948]	0.968	[0.960,0.974]

Table 8: Generality for traffic generation models. The Pearson correlation ρ is used to assess the performance of DeepQueueNet on End-to-End delay/jitter prediction.

		avgRTT		p99RTT		avgJitter		p99Jitter	
		ρ	95% CI	ρ	95% CI	ρ	95% CI	ρ	95% CI
DeepQueueNet	Line4	0.9999	[0.9999,0.9999]	0.9974	[0.9966,0.9980]	0.9985	[0.9980,0.9989]	0.9999	[0.9999,0.9999]
	Line6	0.9999	[0.9999,0.9999]	0.9973	[0.9965,0.9979]	0.9994	[0.9991,0.9995]	0.9999	[0.9999,0.9999]
	Abilene	0.9947	-	0.9920	-	0.9923	-	0.9991	-
	GÉANT	0.9997	-	0.9931	-	0.9991	-	0.9923	-
	2dTorus(4x4)	0.9999	[0.9999,0.9999]	0.9899	[0.9886,0.9911]	0.9949	[0.9942,0.9955]	0.9882	[0.9862,0.9899]
	2dTorus(6x6)	0.9999	[0.9999,0.9999]	0.9912	[0.9906,0.9917]	0.9934	[0.9930,0.9937]	0.9851	[0.9841,0.9860]
	FatTree16	0.9993	[0.9991,0.9994]	0.9879	[0.9854,0.9900]	0.9845	[0.9812,0.9871]	0.9778	[0.9732,0.9815]
	FatTree64	0.9984	[0.9983,0.9984]	0.9765	[0.9751,0.9778]	0.9739	[0.9724,0.9753]	0.9577	[0.9552,0.9600]
	FatTree128	0.9968	[0.9966,0.9970]	0.9572	[0.9546,0.9597]	0.9584	[0.9562,0.9606]	0.9201	[0.9159,0.9243]

Table 9: Topology Generality in the baseline configuration - FIFO + Poisson. The Pearson correlation ρ is used to assess the performance of DeepQueueNet on End-to-End delay/jitter prediction.

		avgRTT		p99RTT		avgJitter		p99Jitter	
		ρ	95% CI	ρ	95% CI	ρ	95% CI	ρ	95% CI
2-class	WFQ	0.989	[0.987,0.991]	0.969	[0.963,0.973]	0.978	[0.975,0.981]	0.931	[0.921,0.939]
	SP	0.989	[0.986,0.991]	0.968	[0.962,0.972]	0.978	[0.975,0.981]	0.930	[0.919,0.939]
3-class	WFQ	0.979	[0.976,0.982]	0.937	[0.927,0.945]	0.962	[0.958,0.966]	0.903	[0.887,0.913]
	SP	0.984	[0.981,0.986]	0.952	[0.943,0.959]	0.971	[0.967,0.974]	0.915	[0.899,0.926]

Table 10: TM Generality. The Pearson correlation ρ is used to assess the performance of DeepQueueNet on End-to-End delay/jitter prediction.

C ADDITIONAL EVALUATION METRICS

In this section, we reveal additional evaluation metrics which are not included in § 6.1 due to space limits. We mainly report two statistical metrics: the Pearson correlation coefficient ρ between the latency distribution of DeepQueueNet and the ground-truth, and the 95% percentile confidence interval (CI) for ρ. ρ is a measure of linear correlation between two sets of data (closer to 1 is better).

Generality for Traffic Generation Models. Table 8 shows a summary of all the experiments we made in Figure 8 We observe that DeepQueueNet's ρ of average RTT is above 0.99 across all scenarios. For 99th percentile (p99) estimations (tail latency), ρ is also above 0.95.

Topology Generality. Table 9 shows the summary of the obtained evaluation results in the baseline configuration coupled with a Poisson arrival process.Across all topologies, DeepQueueNet's ρ is always above 0.99 in terms of average RTT estimation.

TM Generality. To demonstrate TM generality of DeepQueueNet, we show the performance results in Figure 10 and Table 10. Its ρ achieves above 0.97 for 2/3-class SP and WFQ in terms of average RTT estimation, and above 0.93 for p99 RTT estimation.