Towards A Workload-Driven Flow Scheduler For Modern Datacenters

by

Mohamed Malek Naouach

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern datacenters run different applications with various communication requirements in terms of bandwidth and deadlines. Of particular interest are deadlines that are driving web-search workloads e.g. when submitting requests to Bing search engine or loading Facebook home page. Serving the submitted requests in a timely fashion relies on meeting the deadlines of the generated scatter/gather flows for each request. The current flow-schedulers are deadline unaware, and they just start flows as soon as they arrive when the bandwidth resource is available.

In this thesis, we present Artemis: a workload-driven flow-scheduler at the end-hosts that learns via reinforcement how to schedule flows to meet their deadlines. The flow-scheduling policy in Artemis is not hard-coded and is instead computed in real-time based on a reinforcement-learning control loop. In Artemis, we model flow-scheduling as a deep reinforcement learning problem, and we use the actor-critic architecture to solve it. Flows in Artemis do not start as soon as they arrive, and a source starts sending a particular flow upon requesting and acquiring a token from the destination node. The token-request is issued by the source node and it exposes the flow's requirements to the destination. At the destination side, Artemis flow-scheduler is a decision-making agent that learns how to serve the awaiting token-requests based on their embedded requirements, using the deep reinforcement learning actor-critic model.

We use two gather workloads to demonstrate (1) Artemis's ability to learn how to schedule deadline flows on its own and (2) its effectiveness to meet deadlines. We compare the performance of Artemis against Earliest Deadline First (EDF), and two other rule-based flow-scheduling policies that, unlike EDF, are aware of both the sizes and the deadlines of the flows: Largest Size Deadline ratio First (LSDF) and Smallest Size Deadline ratio First (SSDF). LSDF schedules arrived flows with largest size deadline ratio first, while LSDF does the inverse logic. Our experimental results show that Artemis flow-scheduler is able to capture the structure of the gather workloads, maps the requirements of the arrived flows to the order at which they need be served and computes a flow-scheduling strategy based on that. Using the first gather workload that has an equal distribution of flows with (size, deadline) pairs that are equal to (350KB, 40ms) and (250KB, 50ms), Artemis met +35.58% more deadlines than EDF, +24.93% more than SSDF, and performed marginally better than LSDF with +4.42%. For the second workload, 60% of flows have a (size, deadline) pair equals to (350KB, 40ms) and 40% flows with (250KB, 50ms), Artemis outperformed all three flows-schedulers, meeting +16.34% more deadlines than the second best SSDF.

Acknowledgements

An opinion without a π is just an onion. I cultivated my onion and I am baking my pie now. A huge thanks goes for all the people who helped me to cultivate my onion and to bake my pie. A special thanks goes to my cool professor Bernard Wong who has been helping me to decorate my pie. A huge hug goes to all my friends in Shoshin: Shoshin was more than a lab for me, it was a culture, a lifestyle and a state of mind.

Dedication

This is dedicated to the ones I love: mom, dad and sister.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Datacenters are emerging as the new computation paradigm and are becoming the de-facto business choice to develop, test, deploy and run large-scale applications with various communication requirements in terms of throughput and latency. The traffic workload they generate is composed of a collection of flows that differ in size, duration and deadlines. Meeting the communication requirements of these collections of flows rely on accommodating the generated traffic at the flow-level.

In this work, we will be studying how to accommodate deadline-oriented traffic that is prevalent today in web-search workloads [49, 45, 24, 23] for example, when submitting a request to Bing search-engine or loading a Facebook home-page or shopping online at Amazon. Each submitted request [28, 45] hits at first a front-end server and gets dispatched recursively to the edge nodes to be processed. The computed results are then collected back at the front-end server and sent to the user. During this process, each submitted request generates a group of scatter/gather flows that needs to finish within a defined budget of latency to serve each request in a timely fashion.

Retaining online customers depends strongly on serving their submitted requests interactively and processing each one of them in the datacenter facility in a timely fashion. This relies on meeting the deadlines of the generated scatter/gather bursts of flows, and the currently inherited flow-schedulers from the Internet legacy are unaware of deadlines [49] and they start flows as soon as the they arrive when the resource is available. To accommodate deadline traffic in datacenters, one line of work [45, 49, 21] proposed to adopt rate-based allocation approaches with respect to the flows' deadlines. For instance, D3 [49] proposed to modify the commodity switches to perform rate allocation for flows based on their sizes and deadlines. D2TCP [45] modulated the congestion window of the

transport protocol using Explicit Congestion Notifications (ECN) and the flow-deadlines to estimate the sending rate. At the packet level, the deadline of a flow is mapped to a priority that got assigned to its packets, and the tighter the deadline is, the higher the packet priority is. One line of work [8, 19] has been leveraging this to prioritize traffic with high-priority packets and to optimize the tail latency, but this provides no guarantees on meeting flow-deadlines. Some might suggest to adopt the Earliest Deadline Fist (EDF) scheduling scheme at the sender side to schedule flow-packets, but this requires to complement the EDF policy with a new rate control mechanism to identify which packet to send next. pFabric [8] suggested to decouple rate control and packet scheduling at the host side, allowing sources to send traffic at line rate and entrusting the network switches to schedule the packets. The switches' hardware in pFabric are modified to sort the arrived packets by their priorities and to drop low-priorities packets if the ingress queue is full. pHost [19] is an end-to-end transport protocol that was built around the key ideas of request-to-send and clear-to-send to deliver traffic at the packet level and it does perform as good as pFabric without modifying the switches' hardware. pHost needs a heavy parameters' tuning, and it can emulate EDF flow-scheduling when the request-to-send's are prioritized at the host side based on flow-deadlines.

1.1 Artemis

Current flow-schedulers are deadline-unaware and they just start flows as soon as they arrive when the resource is available. Given the recent advancements in machine learning [37, 20, 38, 41, 39], we propose in this work to build a flow-scheduler that is driven by the workload's communication-requirements and learns how to schedule the arrived flows using a reinforcement-learning loop.

In this thesis, we present Artemis: a workload-driven flow-scheduling system for modern datacenters that learns how to schedule deadline-flows via reinforcement. Traffic flows in Artemis do not start as soon as they arrive and each source is required to acquire a token from the flow-destination side to start a particular flow, sending the flow only when the token is acquired. The flow-token request is generated by the source node and it exposes the application-flow requirements to the destination node. At the destination side, Artemis flow-scheduler picks which flow-request to schedule first using the deep reinforcement actor-critic learning model. The applications' communication-requirements are a priori unknown to Artemis and are only exposed to the flow-scheduler at execution-time. Artemis does not initially commit to any fixed hard-coded flow-scheduling policy, and instead, computes the policy on its own in real-time.

1.2 Contributions

In Artemis, we model flow-scheduling as a deep reinforcement learning problem, and we solve it at the end-hosts using the actor-critic architecture. We evaluate Artemis's flow-scheduling system using two specific deadline-driven workloads and we compare its behavior and performance to the fundamental deadline-oriented flow-scheduler: Earliest Deadline First (EDF), and two additional rule-based schedulers that are aware of the sizes and the deadlines of the flows: Small Size Deadline ratio First (SSDF) and Large Size Deadline ratio First (LSDF). SSDF schedules flows with smallest size-deadline ratio first, while LSDF is performing the inverse logic, and schedules flows with largest size-deadline ratio first. We show that:

- Artemis is able to learn how to schedule deadlines flows on its own via reinforcement, starting initially with no prior knowledge about the workload characteristics and using the deep reinforcement actor-critic learning model.
- Artemis is able to capture the workload structure and it maps the requirements of the awaiting flows to the order at which they need to be served to meet their deadlines.
- To demonstrate the ability of Artemis to learn how to schedule flows, we use a gather workload that is evenly composed of flows with (size, deadline) pairs that are equal to (350KB, 40ms) and (250KB, 50ms). Artemis met +35.58% more deadlines than EDF, +24.93% more than SSDF, and performed marginally better than LSDF with +4.42%. Earliest Deadline First fell short because it is intrinsically designed to only consider the flow-deadlines while scheduling and disregard other parameters such as the flow-size.
- To demonstrate the ability of Artemis to adapt its flow-scheduling strategy, we vary the previous gather workload to have 60% of flows with (size, deadline) pairs equals to (350KB, 40ms) and 40% of them with (size, deadline) pairs equal to (250KB, 50ms). Artemis outmatched all three flows-schedulers meeting +16.4% more deadlines than the second best SSDF.

Chapter 2

Background and Related Works

Modern datacenters run different applications with various communication requirements in terms of bandwidth and deadlines. For instance, web-search workloads [45] are deadline driven and their flows need to finish within a defined latency budget, whereas MapReduce workloads [17] are bandwidth aggressive and their flows need to quickly finish. In this work, we research how to accommodate deadline-oriented traffic and in this chapter, we study the state of the art for scheduling deadline flows in a datacenter environment.

2.1 Traffic Delivery In the Era of Datacenters

Today's datacenter networks have inherited from the Internet network-stack legacy essentially two transport protocols that are oblivious to the applications' communication requirements. The first protocol is User Datagram Protocol (UDP) and the second is Transmission Control Protocol (TCP), and neither of them was originally designed to accommodate the applications' traffic at the flow level. In fact, TCP is actually the only protocol that is performing traffic scheduling, and it is limited to the packet level granularity and not higher than that. Unlike the Internet, datacenter networks are expected to operate at a high-speed e.g. 10/40/100 Gbps, and different variants of TCP have been proposed to speed-up the congestion-control feedback-loop and to optimize the finite-state machine of the protocol in order to deliver a high-throughput and an ultra-low-latency. Among all, the most popular proposed variant based on the number of citations is DCTCP [6], which has been recently standardized by IETF [26]. A recent line of work e.g. TIMELY [31] and DX [27], have shown how to outperform DCTCP by leveraging the recent advancements in

Network Interface Controller (NIC) hardware and accurately measure the round trip times (RTTs) by time-stamping packets when sending to sketch a better rate-control system.

Different efforts have been made to replace the conventional UDP and TCP protocols with better-suited transport protocols for datacenter networks that are designed to maximize the throughput and minimize the tail latency. The state-of-the-art research work is twofold: the first line of works [6, 34, 8, 19] target to optimize the datacenter network metrics and the second line of works [45, 21, 33] address to accommodate the applications' traffic at the flow level. In this work, we focus on serving deadline flows and we target to optimize the flow deadline hit ratio metric. In the next sections, we study the traffic characteristics and the state-of-the-art for scheduling deadline-driven workloads in a datacenter network environment.

2.2 Datacenter Workload and Traffic Characteristics

The datacenter network workload is tied to the communication logic of the applications running on the servers and the adopted architectural patterns when deploying services in the facility. For instance, inside the Facebook social media datacenter [36], each server machine has a single unique role out of the following five roles: a web server, a database server, a cache server, a Hadoop server and a multifeed server. Each server is generating and serving a specific type of traffic with respect to its role, and the overall traffic flows in the facility are bursty [1]. The Hadoop traffic-flows are short-lived and small-sized where 70% of flows send less than 10KB and last less than 10 seconds; the median flows send less than 1KB and last less than a second, and in addition, less than 5% of the flows are larger than 1MB and last longer than 100 seconds without exceeding 10 minutes. Due to the use of connection pooling, cache traffic-flows are long-lived and are significantly larger in size compared to Hadoop flows, where 30% of the flows last less than 100 seconds and send less than 1KB, and more than 50% of them exceed 300 seconds, sending less than 30KB. Compared to both Hadoop and cache traffics, web traffic-flows lie somewhere in the middle where the median flows send less than 2KB and last less than 700ms. Facebook provides access upon request to packet traces [18] of three production clusters in the Altoona datacenter [9]. Compared to the observations of Microsoft Bing [24] that are running similar services in concept, Facebook reports similar low link utilization and scatter/gather style traffic patterns, but different load distribution across the facility, where

[1] A burst is a group of consecutive packets with shorter inter-packet gaps than packets arriving before or after the burst of packets. Burstiness is a characterization of bursts in a flow over a period of time. [25, 47]

Table 2.1: The characteristics of median flows inside the Facebook datacenters for different workloads [36]

Median Flows			
Flow Parameter	Hadoop	Web	Cache
Size	$<$ 1KB	$<$ 2KB	$<$ 30KB
Duration	$<$ 1sec	$<$ 700ms	$>$ 300secs

Facebook's distribution appears to be evener. Google [42] also confirms the observation of low link utilization in their fabrics and despite their large capacities, reports that their networks start to experience high congestion drops as link utilization approaches 25%. At the global scale, Statista.com [43] reports that the worldwide datacenter workload is distributed over (computation, video streaming, social networking, search) as (17.14%, 2.86%, 5.71%, 5.71%) and is projected to be (20.43%, 9.68%, 6.45%, 4.30%) by 2020.

Traffic flow characteristics derive from the communication patterns and the type of the traffic generated by the running applications on top of the servers. As shown in figure 2.1, applications' flows vary to be small or large in size, short or long in duration, and with or without deadlines. Previous traffic analysis [12, 13] of ten datacenters that belong to three different business categories (university, enterprise and private cloud) show that overall, 80% of the packets are generated by 20% of the flows, and this observation follows the Pareto principle [48]. Traffic flows could be with or without deadlines. For example, Online Data Intensive (OLDI) applications [28] are expected to serve the users' requests in a timely fashion and requires to be responsive and accommodate a given user request with a defined latency budget: online users can not be kept waiting behind their screens forever to get their requests served, and this correlates positively with the generated revenues [40]. Processing wise, and in the context of web search applications, OLDI applications use a tree-based search algorithms that implements the scatter/gather pattern (a.k.a. partition/aggregate) to process each arrived request. As depicted in figure 2.2, each submitted user-request hits at first a front-end server that dispatches it to a large collection of edge nodes via intermediate servers. Each edge node processes the request and then forwards the results recursively to a final aggregator server that sends it back to the user. During this process, and at each stage, traffic flows are generated in bursts, where a flow has a deadline to meet at each stage (figure 2.3) so that the user request is finally served within a defined latency budget.

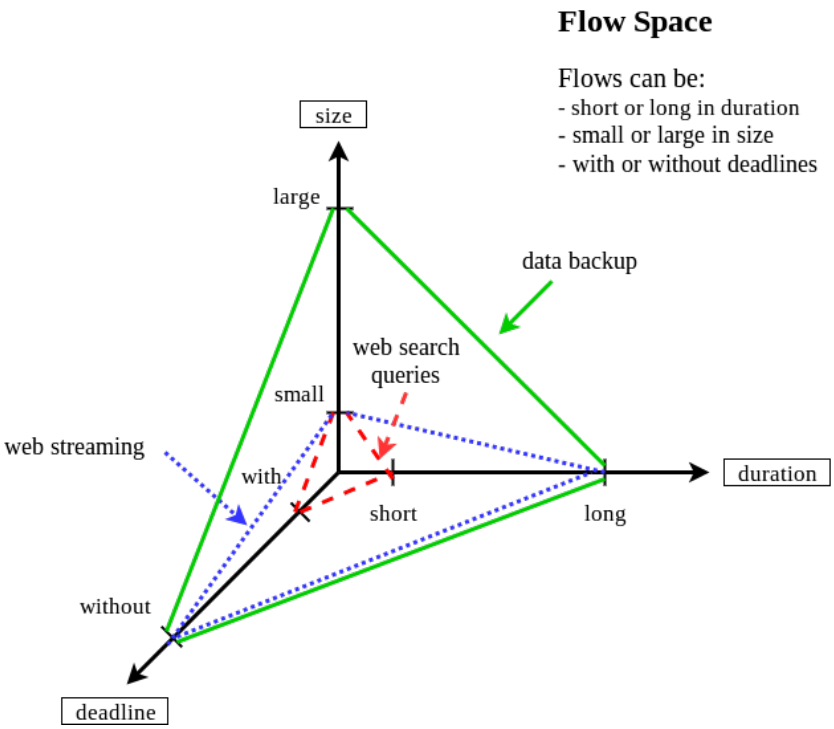

Figure 2.1: Traffic flow space

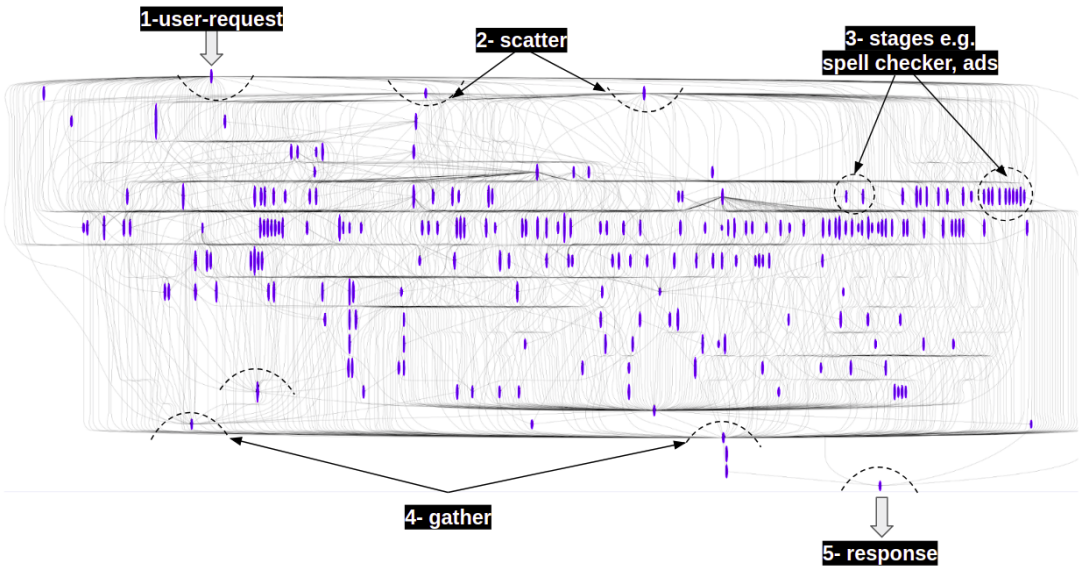

Figure 2.2: Scatter/gather traffic patterns to serve one user request in Bing [23]

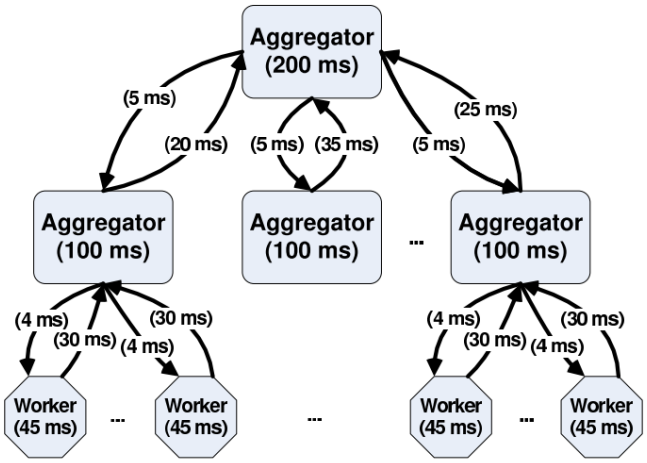

Figure 2.3: An example of the scatter/gather model with associated component deadlines [49]

2.3 Scheduling Deadline Flows

Deadline flows are short in duration and usually small in size. In this paragraph, we study the state of the art for scheduling small-sized flows in datacenters, with and without explicit-defined deadlines.

Scheduling Small-Sized and Deadline Flows At the Host Side

Being completely orthogonal to flow-scheduling, DCTCP [6] relies on controlling the sending rate at the end-host using ECN flags [35] to maintain low queue occupation across the datacenter's switches and hence, reduces the average flow completion time. DCTCP is by default oblivious to the traffic requirements at the flow level and requires to be complemented to be effective for flow-scheduling. For example, to minimize flow completion time, PIAS [10] was built and tested over DCTCP to emulate Shortest Size First (SSF) when scheduling flows without necessary knowing their sizes apriori. PIAS is using a multi-level queue at the sender side to demote the packet-priorities of active flows from high to low in function of the sent bytes, and this results in giving an advantage for small flows to finish before large flows. To accommodate deadline-oriented traffic, D2TCP [45] extend the DCTCP proposal to be aware to the flows' deadlines when adjusting the sending rate in order to reduce the number of missed deadlines. The key idea of D2TCP is to modulate the congestion window size in function of ECN flags and flow deadlines via using the gamma-correction function to back-off aggressively for far-deadline flows and only a little for near-deadline flows. pHost is an end-to-end datacenter transport protocol that is built around the idea of request-to-send and clear-to-send to deliver traffic at the packet level. pHost aims to decouple packet scheduling policies from the core network and performs them instead at the end-hosts. Moreover, the protocol can emulate EDF or SSF scheduling schemes when the issued tokens are assigned to the packets that are part of the flow with the earliest deadline or the shortest size respectively .

Scheduling Small-Sized and Deadline Flows At the Core Network

Small flows live inside the core network for a short period of time, and scheduling them within the network fabric requires the introduction or the modification of the existing hardware. FastPass [34] investigated the idea of introducing a specialized hardware inside the datacenter facility to play the role of the centralized arbiter that is delegated to schedule each packet in the system end-to-end. While FastPass is able to maintain low queue occupancy, it introduces an extra delay for each packet while queuing at the arbiter to get

served, and this risks to negatively affect the average flow completion time and particularly for small flows. In addition, the solution has clear system design limitations like scalability and single point of failure. pFabric [8] proposed to decouple packet scheduling from rate control at the host side, and completely delegate the scheduling task to the network fabric while allowing hosts to start sending at line rate and operate with a minimal rate control. pFabric requires to modify the switches' hardware to sort the arrived packets with respect to their priorities, and to drop low priority packets in favor of high priority packets when the ingress queue is full. The pFabric scheme is targeting to reduce the flow completion time and it is very aggressive against large flows compared to small flows.

Chapter 3

Artemis Flow Scheduling System

In this chapter, we present Artemis: an end-to-end token-based workload-driven flow-scheduling system. Flows in Artemis start sending their traffic upon requesting and acquiring tokens from their destination nodes. At the destination side, Artemis flow-scheduler learns to schedule deadline flows to start based on the requirements embedded in their issued requests. Artemis starts initially with zero-knowledge about the workload characteristics and learns via reinforcement how to schedule deadline flows following the actor-critic learning model.

3.1 Artemis Overview

In Artemis, we adopt an end-to-end approach to schedule the datacenter traffic flows at the destination side. Each source node needs to acquire a token from the destination node it is soliciting to start sending its packets. The token request is issued by the source and it does expose to the destination side the requirements of the flow to start e.g. flow-size, flow-deadline. This design does offer more control over which flows to start at the destination side and gives more flexibility for a source application to communicate their flows' requirements to the destination to be better-served and accommodated.

At the destination side, Artemis flow-scheduler is a scheduler-agent that learns via reinforcement how to schedule deadline flows. We model the flow-scheduling problem as a deep reinforcement learning task and we solve it using the actor-critic architecture. Artemis flow-scheduler adopts initially no particular flow-scheduling strategy. Instead, it just counts on interacting with the environment flows by observing the requirements of the

arrived token-requests, and learns via its feedback-control mechanism how to schedule flows on-the-go. Artemis computes its flow-scheduling policy following the deep reinforcement actor-critic learning model. When a scheduled flow meet its deadline, Artemis will get a reward equals to one otherwise it gets by default a reward equals to zero. During the course of serving latency sensitive traffic, the objective of Artemis is to learn a flow-scheduling distribution over the set of awaiting flows to identify which flow is most likely going to meet its deadline if scheduled first, maximizing the number of the flows meeting their deadlines in the long run.

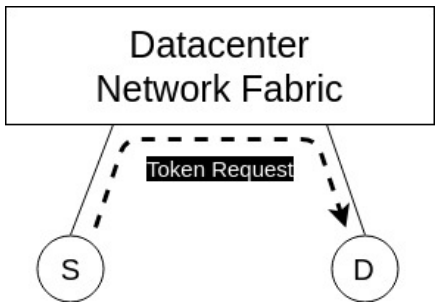

Figure 3.1: In this example, a source node S issues a token-request to the destination node D to acquire a token to start sending its flow. The issued token-request exposes to the destination information such as flow-size, flow-deadline, source load. At the destination side, Artemis flow-scheduler is scheduling which flow to start first based on the embedded pieces of information of the arrived token-requests.

3.2 Artemis Design Objectives

The Artemis flow-scheduling system is designed to be:

- **End-to-End**: Artemis flows' scheduling decisions are computed at the destination side. A source node is allowed to start circulating a particular application-flow only after requesting and acquiring a token from the destination node. At the destination side, Artemis learns to schedule which token-request to serve first, so that its attached flow starts sending its packets.
- **Expressive**: Artemis allows applications running at the source side to express the requirements of the flows they are going to start and expose them in their issued token-requests. Flow requirements vary in terms of bandwidth and deadlines and

could be expressed for example in function of the deadlines and min/max-rates. In this work, we study how to schedule deadline-oriented traffic, and each issued token-request is exposing the size and the deadline of the flow to the destination side.

- **Workload-Driven**: Artemis is not a rule-based flow-scheduling system. Instead, it is an agent that learns its flow-scheduling policy via reinforcement by interacting with the system workload. The flow-scheduling policy computed by Artemis is driven by the requirements of the arrived flows that are waiting to be started. Artemis starts initially with zero prior knowledge about the workload, and computes a probabilistic flow-scheduling strategy following the deep reinforcement actor-critic learning model.

3.3 Flow-Scheduling as a Deep Reinforcement Learning Problem in Artemis

In Artemis, we adopt a learning-based approach where we teach the flow-scheduler how to interact with the environment-traffic and accommodate the arrived flows. The teaching process is using a feedback-control mechanism to evaluate how good the flow-scheduler is performing in terms of accommodating the applications' traffic requirements. The adopted learning paradigm is known in the literature as reinforcement learning [44], and Artemis flow-scheduler is designated as a reinforcement learning agent.

As illustrated in figure 3.2, a reinforcement learning agent A interacts with an environment E, and at each time step t, it observes the environment's state s_t, takes an action a_t and receives a reward r_t. The agent's actions are determined by a stochastic policy π that is updated iteratively at each time step: the policy π is a probability distribution function over the set of actions a's to pick from at a given state s, based on its contribution to the long-term cumulative-reward value calculated using a state-value function $V(s)$ a.k.a. value function. The performed action a_t at each time step t is not necessary always maximizing the expected cumulative reward, and a reinforcement learning agent is allowed to explore the complete set of possible actions: this is known in the literature as the principle of exploration-exploitation [44]. The action decision process is Markovian, and the next action a_t to take only depends on the environment's current state s_t, and ignores all previous states. As presented in figure 3.3, we define these components in Artemis as follows:

- **Environment** E: The environment is what defines the world that Artemis flow-scheduler is interacting with. It produces a state and a reward for the flow-scheduler

agent to observe and to process consecutively, and it accepts an action from the agent and cycles back to produce another state again. The environment is composed of the different source-nodes issuing token-requests to start their flows. These issued requests queue up in the natural arrival queue of the solicited destination-node, and the embedded requirements of the first f awaiting requests (f is a system parameter to configure) define the state s Artemis flow-scheduler is continuously observing to make a decision on which request to serve next. When the token-request is served, its attached flow is scheduled to start, and if the flow finishes within its deadline, Artemis flow-scheduler gets a positive reward equal to 1, else a zero.

- **Agent** A: The agent learns how to achieve a defined goal by interacting with the environment. At each time step, Artemis flow-scheduler observes the requirements of the first f awaiting token-requests at the front of the system queue if any, and decides which token-request f_i to serve and which token-request f_i to defer, where $1 \leq i \leq f$. If the token-request f_i is served, the attached flow to f_i is scheduled to start. To limit the competition among the awaiting token-requests over the destination bandwidth-resource, we limit Artemis flow-scheduler agent to only serve k token-requests at a time so that only k flows are concurrently active. Both f and k parameters are manually tuned in Artemis and could be also integrated into the learning process. For instance, k should be larger in 40Gbps networks compared to its value in 10Gbps networks, and as for the parameter f, when the traffic is highly variable in requirements, f should to be set to a large value so that the reinforcement learning agent can catch more insights about the traffic's structure to learn a better suited flow-scheduling policy.

- **State** s: The requirements of the f awaiting token-requests at the front of the system queue define the state s for Artemis flow-scheduler agent. Each token-request f_i exposes the size and the deadline of the flow.

- **Action** a: At each time step t, Artemis decides which token-request f_i to serve out of the f awaiting requests at the front of the system queue, where $1 \leq i \leq f$.

- **Reward Function** r: The reward is the feedback by which the flow-scheduler agent measures the success or the failure of its taken action. We will be evaluating Artemis using deadline-driven workloads, and we define the reward function to generate either 1 or 0. If the scheduled flow finishes within its deadline, Artemis flow-scheduler agent receives a reward equals to 1, otherwise it is a zero. Started flows do not finish immediately, and this makes rewards lagged in Artemis. To solve this problem, Artemis implements a reward-buffer queue to collect the rewards of the the finished

flows asynchronously, and to assign them subsequently to the next performed actions. In the absence of an immediate reward to assign for a performed action, Artemis adopts a pessimistic approach and gets by default a reward equals to zero.

- **Policy** $\pi(a|s)$: The policy is the strategy that the agent follows to perform a scheduling-action a at state s. At each time step t, a token-request f_i is either served or deferred. With f token-requests defining the flow-scheduler state s, the number of possible policies to try and evaluate is 2^f: a policy π is better than a policy π' if it is successfully accommodating a higher number of deadline flows in the long run. The requirements of each flow in Artemis are defined by the flow-size and the flow-deadline, and each one of them could possibly have a wide range of values: this makes the size of potentially observed states by the flow-scheduler agent very large, and it would result in an exponential number of policies that are impossible to store in a look-up table. We use therefore a function approximator [15, 29] to estimate which action a to take at a given state s and iteratively compute the policy π to adopt. The estimation is calculated using a deep neural network represented by the vector of weights θ, and the policy π is therefore denoted by $\pi(a|s;\theta)$.

- **State-Value Function** $V_\pi(s)$: As opposed to the 0 and 1 rewards collected immediately after finishing sending a particular flow, the value function $V_\pi(s)$ estimates the number of flows to meet their deadlines during the exercise of scheduling n flows, starting from state s and following policy π. The state-value function is used to evaluate the quality of the current policy π and to improve it iteratively following the actor-critic learning model. Artemis flow-scheduler is learning a flow-scheduling policy π to identify which flow is more likely to meet its deadline if scheduled at state s. In other words, and at a state s, the objective of Artemis flow-scheduler is to act with action a and schedule the flow with the best return value. The flow scheduling is a continuous control process, the goal of Artemis is to compute a flow-scheduling policy that is rewarding in the long-run. That's why, we will be using a discount factor γ to attenuate the effect of the short-term returned rewards.

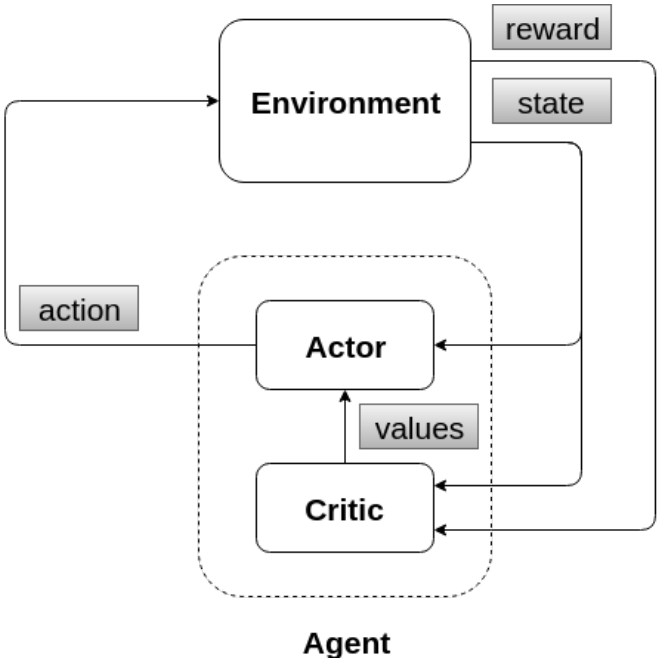

Figure 3.2: Actor-critic reinforcement learning problem[44]

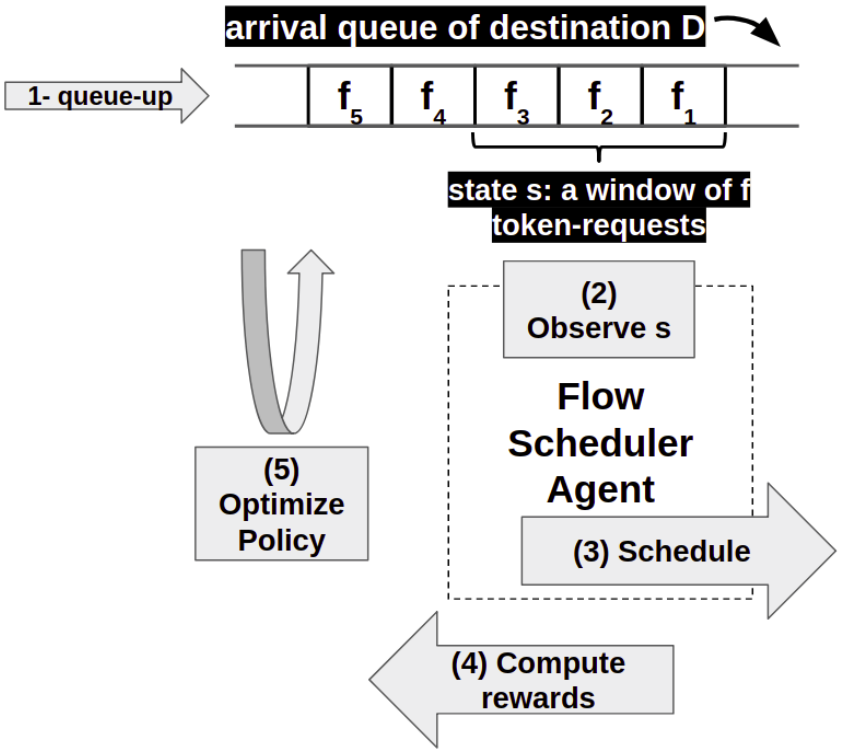

Figure 3.3: Scheduling flows in Artemis using reinforcement learning

3.4 Artemis Architecture

Artemis solves the flow-scheduling deep reinforcement learning task using the actor-critic architecture. As shown in figure 3.4, at each time step t, the actor acts out an action a_t based on the current calculated flow-scheduling policy, and keeps on improving it iteratively based on the evaluations provided by the critic. Artemis flow-scheduler is an asynchronous actor-critic agent [32] at the destination side that uses a neural network as a function approximator to estimate which flow-scheduling action a_t to perform at state s_t. To regularize the flow-scheduling learning process, we use the same neural network to evaluate the quality of the performed actions by Artemis.

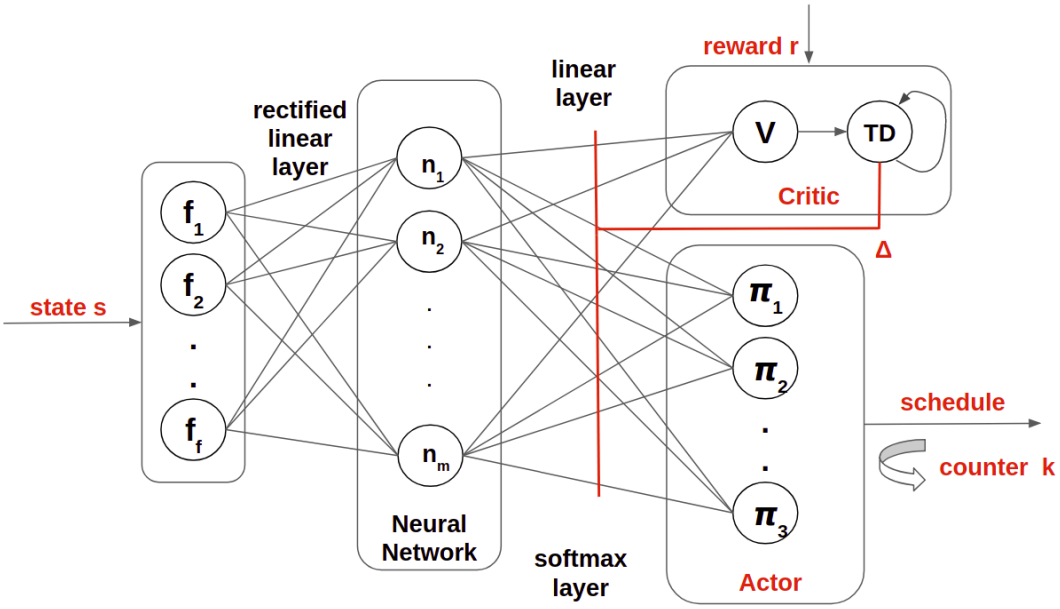

Figure 3.4: Artemis flow-scheduler agent in one figure

3.4.1 System Design

Artemis flow-scheduler is designed to operate as summarized in algorithm 1. The issued token-requests by the different sources across the facility queue up at the natural queue of the designated destination. At each time step t, the flow-scheduler observes the first f requests at the front of the queue and using the current computed flow-scheduling strategy π, serve one request f_i at a time while maintaining a counter k to keep track of the number of the concurrent active flows at the destination side to not to exceed a maximum value. when a flow-request f_i is served, its attached flow is scheduled to start, and by the time the flow finishes transmitting its packets, Artemis gets a positive reward equals to 1 if the flow-deadline is met, otherwise, it is a zero. The f and k parameters depend on the composition and the characteristics of the traffic flows, including and not only restricted to the degree of traffic convolution and the variability of flows' requirements. We therefore treat both of them as tuning parameters in the present design, and we manually configure

them.

```
while arrived flow-bursts do
    if just starting then
        Get the initial set of requirements of the f awaiting flows at the front queue;
    end
    if finished flow then
        Decrement k;
        Buffer the collected reward r in the flow-scheduler reward queue;
    end
    if # of scheduled flows < k then
        Schedule the next flow to start;
        Increment k;
        Get the reward r and update the set of f-requirements;
        Train the flow-scheduler agent;
    end
end
```

Algorithm 1: Pseudocode for the interaction between Artemis flow-scheduler and the environment

3.4.2 Learning Process

At its core, Artemis flow-scheduler is a neural network to decide on which flow to schedule at each state s and to evaluate the quality of the performed actions. The input size of the neural network depends on the the number of flow-requests to observe at state s. As for its architecture, it depends on the complexity of the traffic, and the degree of convolution of the embedded pieces of information that are exposed in the arrived token-requests.

Artemis flow-scheduler is learning how to schedule flows based on the actor-critic duality. As summarized in algorithm 2, the actor is responsible for computing and optimizing the stochastic policy $\pi_\theta(a|s)$ that outputs a probability distribution over the set of actions at each state s. At each time step t, and when making the decision about the next token-request to serve, the actor is exploiting the gained knowledge it has been accumulating with a probability equals to $1 - \epsilon$, and exploring with a probability ϵ a new set of actions

following a uniform distribution.

Draw a probability p using a pseudo random generator;
if $p < \epsilon$ **then**
  Select a flow f_i to schedule using the uniform distribution;
else
  Select a flow f_i to schedule based on the current policy learned by the actor;
end
Reduce ϵ linearly to exploit the gained knowledge;

Algorithm 2: Pseudocode of the decision making process of the actor component

The actor is optimizing its stochastic policy $\pi_\theta(a|s)$ using the evaluations of the critic (algorithm 3). The quality of each action a is evaluated in relation with the next n following rewards and discounted with a γ factor to attenuate the effect of short-term rewards.

while *agent is learning* **do**
  Schedule the flows using the current policy π_θ;
  Collect a sequence of states s_t, actions a_t, rewards r_t;
  Aggregate each sequence of length n into one training point considering the discount factor γ;
  Evaluate the quality of action a_t at state s_t and calculate the advantage;
  Perform gradient ascent to optimize the policy π_θ of the actor;
end

Algorithm 3: Pseudocode of Artemis flow-scheduler learning algorithm

Chapter 4

Evaluation

We implemented Artemis using ns-3 [2] and tensorflow [1]. In this section, we evaluate Artemis using gather workloads composed of flows with (size, deadline) pairs equal to (350KB, 40ms) and (250KB, 50ms) with a 50-50 split in section 4.2 and with a 60-40 split in section 4.3. We demonstrate the capabilities of Artemis flow-scheduler agent and we compare its performance against the universal deadline-aware flow scheduler EDF and two size/deadline ratio-aware flow schedulers: Smallest Size Deadline ratio First (SSDF) and Largest Size Deadline ratio First (LSDF). Using variant 1 and 2 of Workload A, we show that Artemis is able to capture the workload structure and learns a flow-scheduling policy that meets +35.58% and +48.34% more deadlines than EDF consecutively, and +4.42% and +16.34% better than the second best LSDF and SSDF respectively.

4.1 Experimental Setup

Artemis flow-scheduler is a learning agent that operates at the end-hosts. We target to evaluate (1) its learning capability and (2) its performance to execute deadline-driven workloads. In this section, we first define our experimental setup and we configure the workloads we are going to use to achieve both mentioned objectives. We set the baseline for Artemis to be the universal deadline-aware flow-scheduler Earliest Deadline First (EDF).

4.1.1 Datacenter Topology

In our conducted experiments, we do not use any particular datacenter network-architecture. Modern datacenter networks have high bisection bandwidth [42], and therefore, we abstract the datacenter core fabric as a big non-blocking switch, where each node has a full-duplex connection to the core network with a link capacity equals to 1Gbps and a link delay equals to 250us.

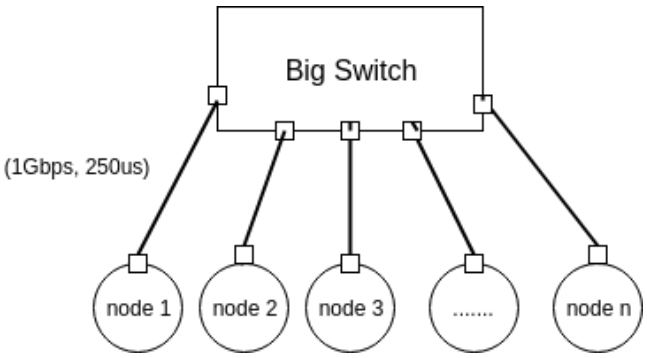

Figure 4.1: Abstraction of the datacenter network as a big non-blocking switch

4.1.2 Methodology and System Configuration

The first objective of this experimental study is to demonstrate the ability of Artemis to schedule deadline flows on its own, without being initially hard-coded to operate with a given flow-scheduling policy. To this end, we will start initially comparing against Earliest Deadline First, and we design a specific gather task workload A (figure 4.2) configured in a way that the scheduling decisions of EDF are not optimal, and this will lead it to fall short in meeting the deadlines of the flows. The reason behind that is to study Artemis's learning-capability and investigate whether or not it is able to learn a better strategy than EDF, given that we know about its existence.

We will be using the gather task workload A (or simply workload A) represented in figure 4.2. Workload A is composed of 10-to-1 gather task flows arriving in bursts that follow a Poisson distribution with an arrival-rate equals to 10 bursts/sec. Each burst is composed of a group of 10 flows that are distributed as follows:

- 5 flows, each one of them has a size of 350KB and a deadline of 40ms.
- 5 flows, each one of them has a size of 250KB and a deadline of 50ms.

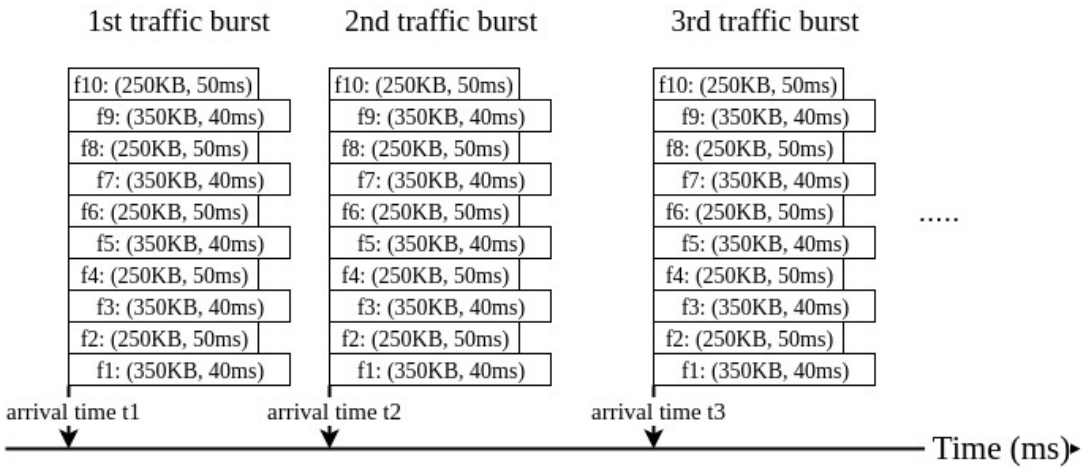

Figure 4.2: Distribution of the gather task workload A (variant 1)

To limit competition between flows at the destination side, we cap the number of concurrent active flow to k. To make it easy to reason about the behaviors of the flow-schedulers, we set the value of k to be equal to the number of arrived flows per burst (which is 10 for workload A). Note that the maximum number of concurrent flows to start does affect the performance of the flow scheduler. Let k denote the maximum number of concurrent flows to start by the flow-scheduler. We vary k exponentially and we execute workload A (variant 1) using the FIFO flow-scheduler. Figure 4.3 shows how the flow completion time (FCT) performance of the flow-scheduler changes in function of k: FIFO performs poorly for small (≤ 3) and large (≥ 63) values of k, where 50% of the flows are taking more than 60ms to finish.

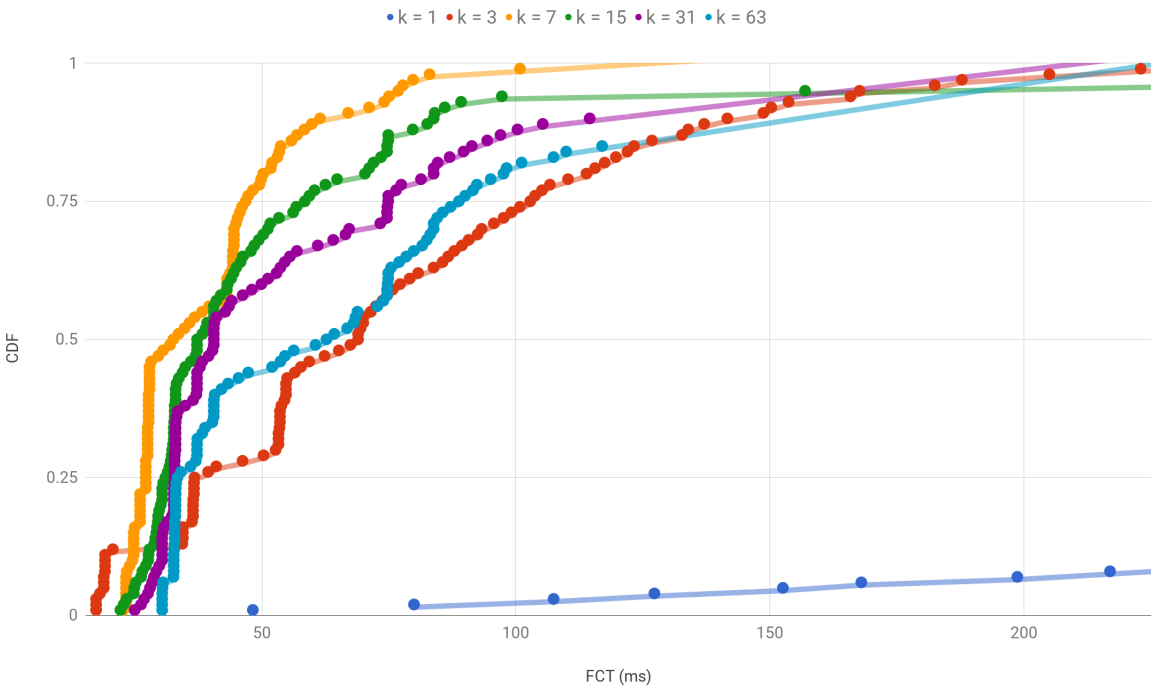

Figure 4.3: Flow completion time in function of the maximum number of concurrent flows to start, FIFO scheduler; workload A (variant 1)

In addition to that, the configuration of the flow-deadlines' values does also affect the flow deadline hit (FDH) ratio performance of the flow-scheduler. Based on the previous flow completion time graph, 50% of the flows were able to finish within 40ms for k's between 7 and 31. If all the workload's flows have a tighter deadline-value (e.g. 30ms), the flow-scheduler will fail to accommodate more or less half of the deadlines with respect to k. All above, we set the values of the flow-deadlines to be moderate (e.g. 40ms) and lax (e.g. 50ms). Similarly, we vary k exponentially and we re-execute workload A (variant 1) using the EDF flow-scheduler this time, and we show how the k parameter does affect the flow deadline hit ratio performance of the flow-scheduler (figure 4.4). We also plot the flow-completion-time cumulative distribution of EDF at figure 4.5. The number f of the awaiting-flows to observe by Artemis and decide on which flow to schedule first does affect the learnability and the performance of the scheduling task, and it will be set during this evaluation to $k/2$, which equals to the number of flows in a half-burst.

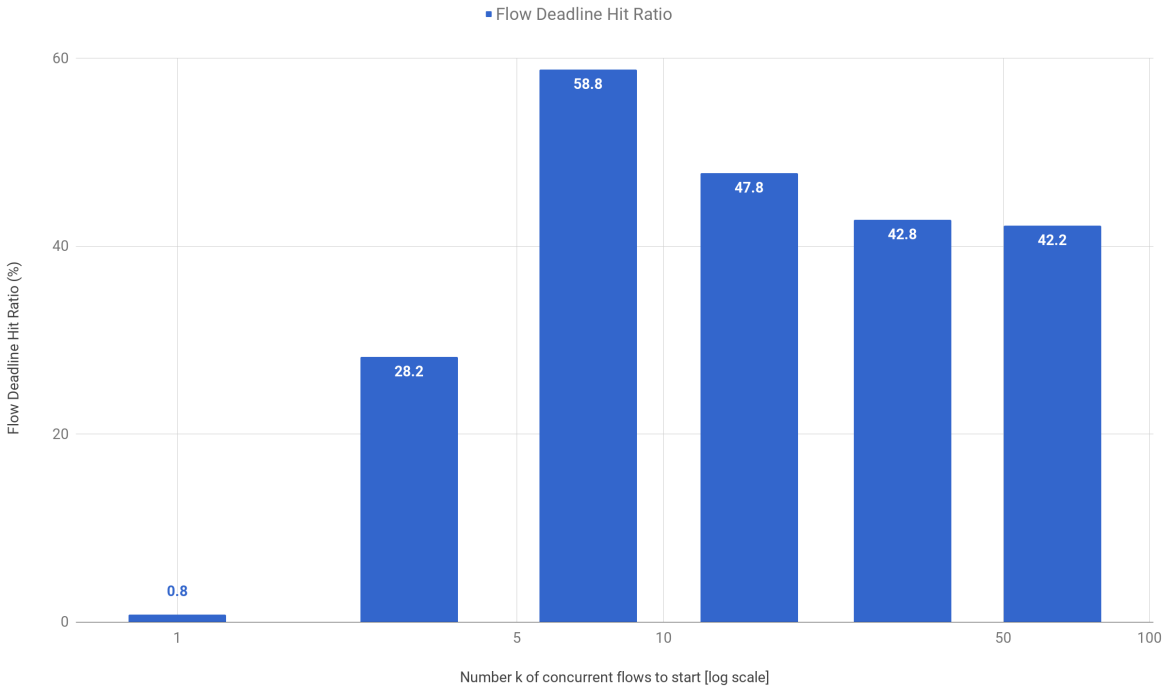

Figure 4.4: Flow deadline hit ratio in function of the maximum number of concurrent flows to start, EDF scheduler, moderate deadlines; workload A (variant 1)

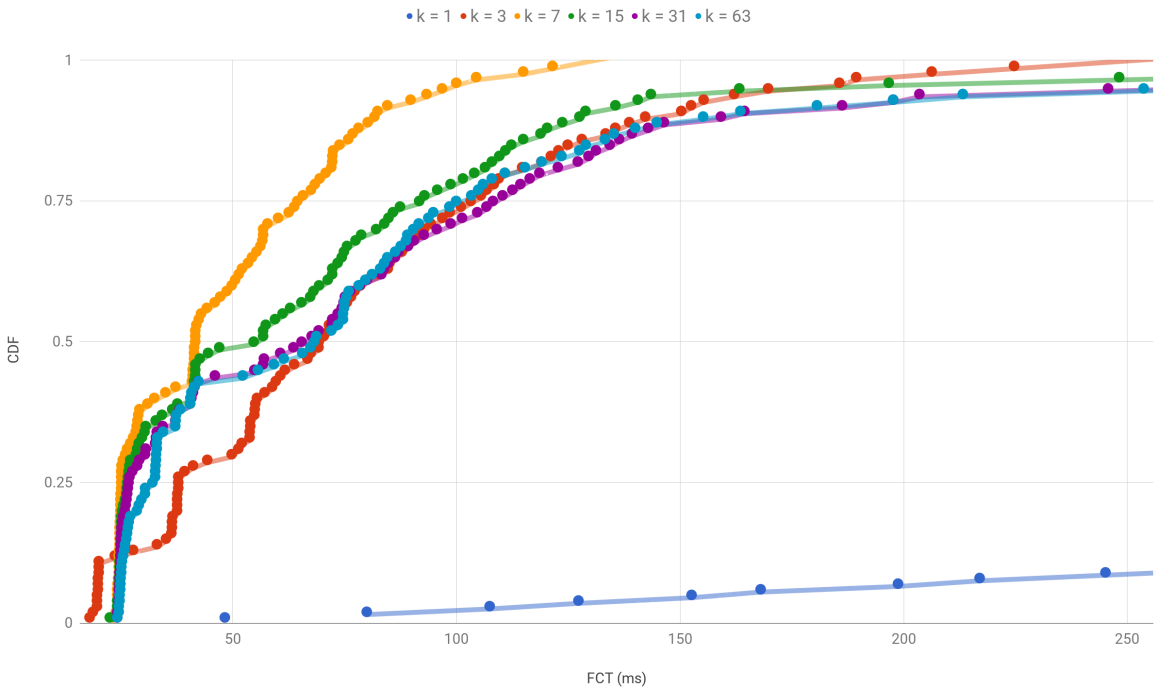

Figure 4.5: Flow completion time in function of the maximum number of concurrent flows to start, EDF scheduler, moderate deadlines; workload A (variant 1)

The actor and the critic entities in Artemis flow-scheduler are using the same neural network to evaluate and optimize the flow-scheduling policy. With respect to the defined workload, we start off with a fairly simple neural network with one layer and 16 neurons, and it was effective to capture the structure of workload A. To evaluate the quality of the flow-scheduling decisions that are performed by the actor, the critic estimates the number of the satisfied deadlines with respect to the n previous performed decisions. In our system, an incorrect performed scheduling decision could make the few next flows miss their deadlines, and the effect of a particular flow-scheduling decision is discounted over time with a γ factor that varies to be between 0 and 1. We set n and γ to be equal to 8 and 0.99.

4.2 Artemis In Action: Workload A (variant 1)

As illustrated in figure 4.2, traffic in workload A (variant 1) arrives in bursts of 10 flows: each burst contains 5 flows with (size, deadline) pairs equal to (350KB, 40ms), and 5 more flows with (size, deadline) pairs equal to (250KB, 50ms). Each of the flows is requesting to acquire a token from a single destination to start sending its packets. At the destination side, we start by evaluating the behavior and the performance of Artemis versus the deadline-aware flow-scheduler Earliest Deadline First. Next, we compare Artemis versus two flow-schedulers that are aware of the sizes and the deadlines of the flows: Smallest Size Deadline ratio First (SSDF) and Largest Size Deadline ratio First (LSDF). SSDF schedules flows with smallest size-deadline ratio first, whereas LSDF performs the inverse logic and schedules flows with largest size-deadline ratio first.

4.2.1 Artemis vs Earliest Deadline First

By definition, Earliest Deadline First's logic cares only about the deadlines of the flows when scheduling. The intrinsic nature of workload A (variant 1) makes the decision of Earliest Deadline First flow scheduler biased towards the 40ms flows and aggressive towards the 50ms flows, regardless that 50ms-flows are lighter in size and have hence higher-chances to meet their deadlines if scheduled first.

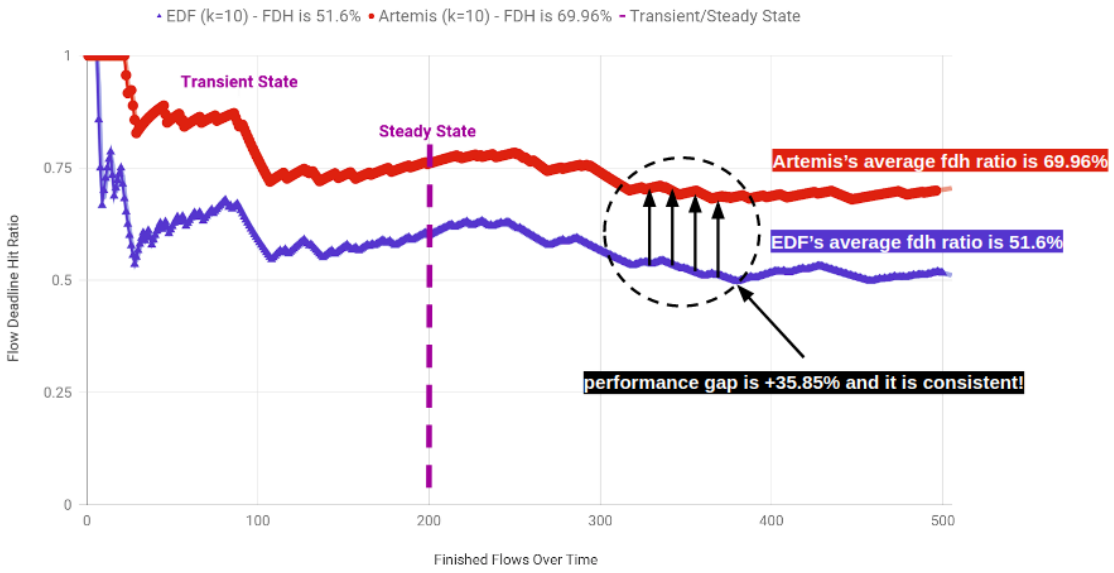

Figure 4.6: Flow deadline hit ratio for Artemis vs EDF; workload A (variant 1)

Flow Deadline Hit Ratio and Flow Completion Time

As shown in figure 4.6, EDF succeeds to meet 51.6% of the deadlines of workload A (variant 1) that is composed of an even split of both 40ms and 50ms-flows. Unlike EDF, Artemis is not myopic to the flows' sizes and that's why it does perform better than EDF, satisfying 69.96% of the deadlines. At the experiment's steady state, Artemis is consistently scoring a better flow deadline hit ratio, with a performance-improvement margin equals to +35%: this demonstrates first how Artemis is able to figure out how to schedule deadlines flows on its own in real-time, and second, how Artemis is reinforcing what it is learning on the long-run. Artemis learned to prioritize the 50ms flows and not to starve the 40ms flows unlike EDF, and that's why it succeeded to accommodate more than 2/3 of the volume of the deadline flows.

During the experiment's steady state, we plot the flow-completion-time cumulative distribution (figure 4.7) for Artemis and EDF. The graph shows that both flow-schedulers have an exponential-shaped fct-cdf, with EDF having the curve with the longer tail. Using EDF, 50% of the flows finished within 43.35ms and 75% of them finished within 75.44ms.

Table 4.1: Volume of flows finishing within the earliest (40ms) and the latest (50ms) deadline for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently		
Flow Scheduling Policy	FCT $\leq$ 40ms	40ms $<$ FCT $\leq$ 50ms
EDF	40%	54-40 = 14%
Artemis	62%	76-62 = 14%
SSDF	54%	71-54 = 17%
LSDF	56%	75-56 = 19%

As for Artemis, its learned flow-scheduling policy resulted in 50% of the flows taking 31.13ms to finish, and 75% of them taking 49.68ms to finish.

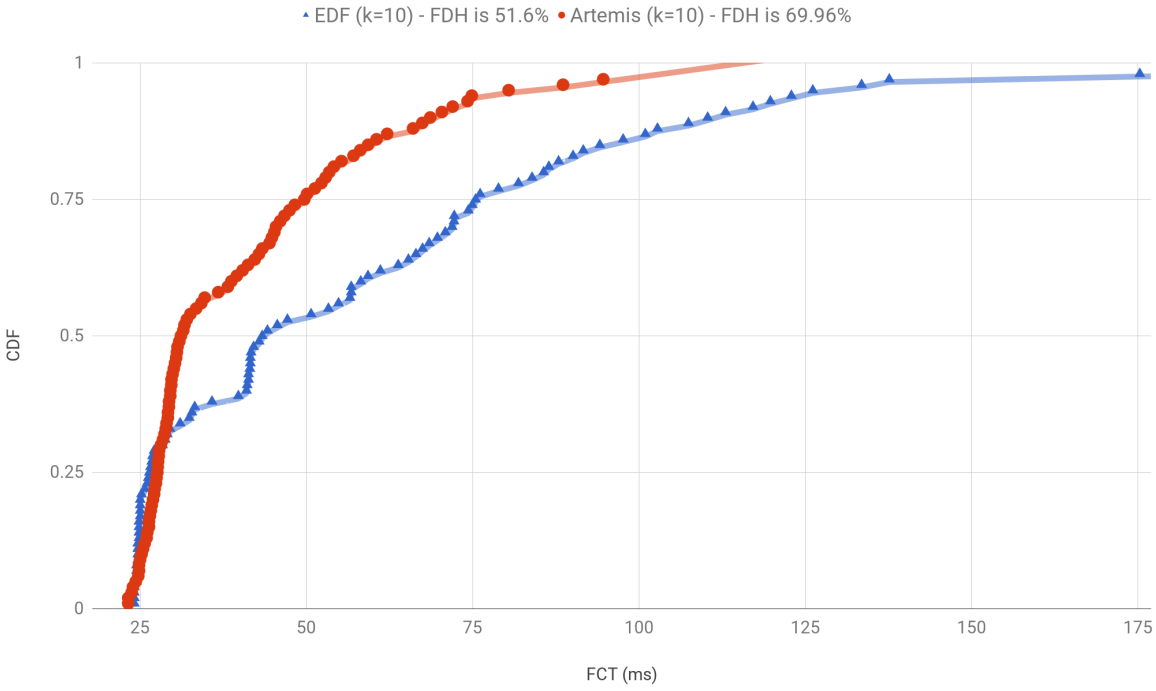

Figure 4.7: Flow completion time cumulative distribution function for Artemis vs EDF; workload A (variant 1)

Table 4.2: Volume of the deadline-missed 40ms-flows to accommodate more 50ms-flows for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently		
Flow Scheduling Policy	Flow Deadline Hit Ratio	volume of non-useful 40ms-flows with FCT < 50ms
EDF	51.6%	53.8-51.6/53.8 = 4.09%
Artemis	69.96%	75.7-69.96/75.7 = 7.58%
SSDF	56%	70.3-56/70.3 = 20.34%
LSDF	67%	74.4-67/74.4 = 9.95%

Table 4.3: Volume of deadline-met 50ms-flows finishing after 40ms for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently		
Flow Scheduling Policy	Flow Deadline Hit Ratio	volume of useful 50ms-flows with FCT > 40ms
EDF	51.6%	51.6-39.11/51.6 = 24.21%
Artemis	69.96%	69.96-61.8/69.9 = 11.67%
SSDF	56%	56-53.97/56 = 3.63%
LSDF	67%	67-55.6/67 = 17.01%

Useful and Non-Useful Deadline Flows

A deadline flow is useful if it meets its deadline. Flow-deadlines in workload A (variant 1) are bimodal-distributed between 40ms and 50ms in an equal proportion. As shown in tables 4.1, 4.2 and 4.3, Artemis flow deadline hit ratio's performance can be basically explained with the following two observations. First, based on table 4.1, Artemis learned how to schedule 62% of the workload flows, allowing them to finish within 40ms. If Artemis was prioritizing the 40ms-flows, it would performs like EDF but, it is not the case here and this is not surprising: when it comes to finishing flows quickly, shorter flows finish before larger flows, and the 40ms-flows (with a 350KB payload) are larger in size compared to the 50ms-flows (with a 250KB payload). Artemis seems to favor the 50ms-flows over the 40ms-flows and schedule them first. Second, Artemis learned to defer the 40ms-flows which are heavier in size than the 50ms-flows and to not always prioritize them like EDF but instead, trade them with the 50ms-flows. In table 4.2, we measured the volume of the 40ms-flows

that missed their deadlines and finished in less than 50ms to accommodate more 50ms-flows, and we found that their volume is equal to 7.58%: Artemis deferred these 40ms-flows to be start later, trading them instead with the 50ms-flows which are lighter in size and hence, are more likely to meet their deadlines. This demonstrates that Artemis was able to identify that the 50ms-flows are more useful than to the 40ms-flows when scheduled first. Compared to Artemis, EDF flow-scheduling logic resulted in a smaller volume of these 40ms-flows that finish is less than 50ms, and their volume is only equal to 4.09%. Table 4.3 shows that 24.21% of the 50ms-flows (out of 50%) finished later than 40ms when EDF is used, and this reflects EDF's logic to always prioritize the 40ms-flows over the 50ms-flows. As for Artemis, the same table shows that only 11.67% of the 50ms-flows (out of 50%) finished later than 40ms, whereas the rest of them should finish within 40ms to achieve a flow deadline hit ratio equals to 69.96%. This demonstrates that Artemis figured out that the 50ms-flows are more likely to meet their deadlines compared to the 40ms-flows, and it will be more useful to prioritize them first to meet as many deadlines as possible.

Flow Service Rate and Destination Idle Time

The inter-flow finish time reflects the system service time and its capacity to accommodate the workload traffic per unit of time. As shown in figure 4.8, both of Artemis and EDF served 50% of the flows almost at a similar rate and what made the difference in performance is how the second half of the flows had been accommodated. Traffic bursts come in groups of 10 flows per burst, such that the average burst-arrival time is equal to 100ms (we verified this experimentally and we recorded 100.15ms), i.e. 10 bursts/sec and hence, 100 flows/sec. As listed in table 4.4, EDF served flows in average at rate 83.178 flows/sec (out of the 100.15 flows/sec arrival rate that was experimentally measured), utilizing the destination link at the ratio 83.046%. Artemis was able to reach a link-utilization ratio equals to 98.843% by learning how to serve flows at rate 98.994 flows/sec.

Table 4.4: Average flow service rate and destination link utilization for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently			
Flow Scheduling Policy	Average Inter-Flow Finish Time (ms)	Average Flow Service Rate (flows/sec)	Average Destination Utilization
EDF	12.023	83.178	83.046%
Artemis	10.101	98.994	98.843%
SSDF	12.025	83.158	83.031%
LSDF	10.502	95.218	95.072%

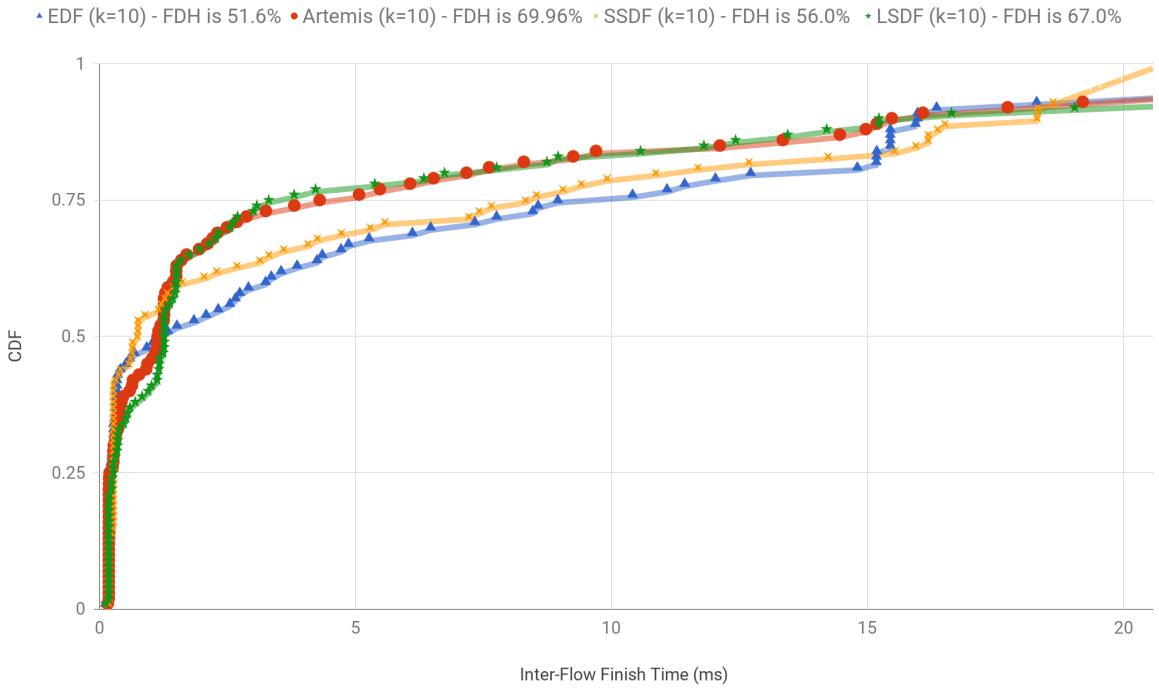

Figure 4.8: Inter-flow finish time cumulative distribution function for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Table 4.5: Mean/median flow completion time and flow deadline hit ratio for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently			
Flow Scheduling Policy	Average Flow Completion Time (ms)	Median Flow Completion Time (ms)	Flow Deadline Hit Ratio
EDF	66.036	45.350	51.60%
Artemis	55.204	31.132	69.96%
SSDF	45.561	38.461	56.00%
LSDF	150.203	34.061	67.00%

Mean/Median Flow Completion Time and Straggler Flows

Nevertheless, even with a high flow-service rate and a high destination-link utilization, we notice that the system is not completely free of flows that straggle to finish either for EDF or for Artemis. To observe this, we compare the average flow-completion-time to its median. As shown in table 4.5, EDF's average flow-completion-time is equal to 66.036ms and this is relatively higher with +45.614% compared its median 45.350ms. As for Artemis, the average flow-completion-time and its median are consecutively equal to 55.204ms and 31.132ms, making the average fct higher with +77.322% to its median. In both cases, the median fct was less than the mean, and this indicates that the fct probability distribution function (PDF) for both flow-schedulers is positively skewed. Hence, each of the flow-scheduler's fct-pdf has a long tail on the right side, and actually, Artemis's tail-latency is shorter and thinner than EDF (figure 4.11).

A straggler flow is a flow that takes more than expected to finish. We note that the near absence of straggler flows in the system does not necessary imply any properties on the flow-scheduler's capabilities to meet deadlines. In the next section, we will see that SSDF has the shortest tail latency among all four flow schedulers, but has only succeeded to meet 56.0% of the deadlines which is slightly better (+8.53%) than EDF and +24.98% behind Artemis. Similarly, the LSDF flow-scheduler generated the heaviest tail latency while meeting 67.0 % of the deadlines.

4.2.2 Artemis vs Smallest and Largest Size-Deadline ratio First

In the previous section, EDF was performing poorly compared to Artemis because it was only considering the deadlines of the flows and not their sizes when scheduling. The nature of the workload fooled EDF to prioritize the 40ms over the 50ms-flows, despite that the 50ms-flows were lighter in size and this makes them more likely to meet their deadlines if scheduled first. In this section, we compare Artemis against two flow-schedulers that are simultaneously aware of the deadlines and the sizes of the flows: Smallest Size Deadline ratio First (SSDF) and Largest Size Deadline ratio First (LSDF). SSDF prioritizes flows with the smallest size/deadline ratio first, while LSDF performs the inverse logic. We show that LSDF is able to accommodate almost as many deadlines as Artemis, meeting 67% of the deadlines versus 69.96% for Artemis.

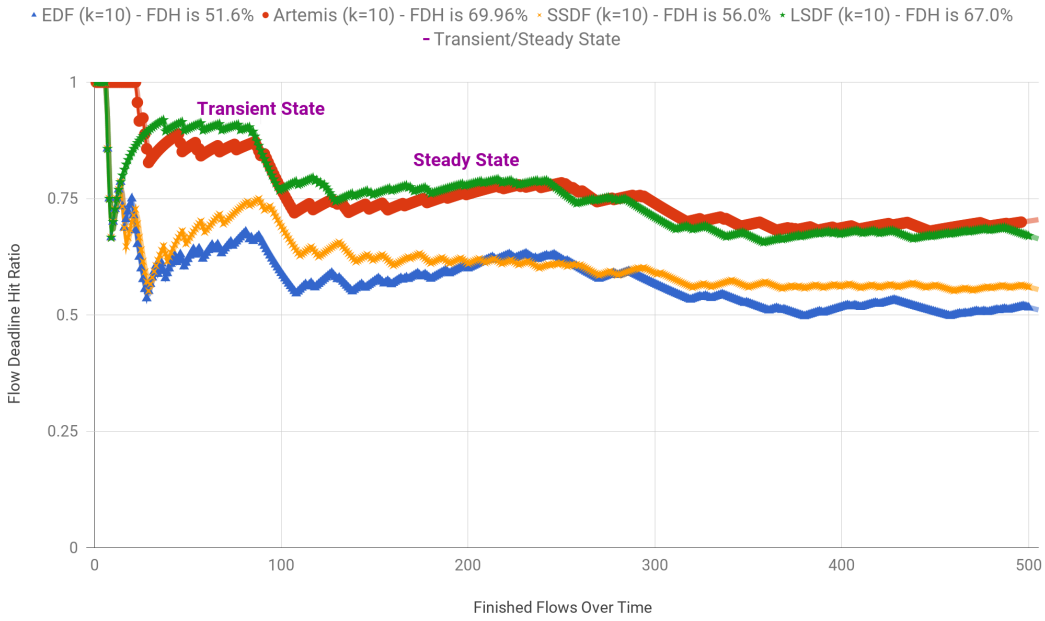

Figure 4.9: Flow deadline hit ratio for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Flow Deadline Hit Ratio and Flow Service Rate

As illustrated in tables 4.4 and 4.5, SSDF's performance is very close to EDF, and it scored respectively 56% vs 51.5% for flow deadline hit ratio and 83.158 flows/sec vs 83.1738 flows/sec for flow service rate. Based on figure 4.8, we note that SSDF is serving the volume of flows between the 50th and 70th percentiles at a higher-rate and this explains its marginal gain ahead of EDF. More importantly, LSDF is quite competitive to Artemis, staying slightly behind it and scoring respectively 67% vs 69.96% for flow deadline hit ratio, and 95.218 flows/sec vs 98.994 flows/sec for flow service rate.

Flow Completion Time and Head/Tail Latency

As shown in figure 4.10, all of the four flow-schedulers have an exponential fct-distribution shape, with LSDF having the longest tail and SSDF having the shortest tail. The fct-cdf's of both of SSDF and LSDF are interleaving with Artemis, following approximately similar growth-rates.

Artemis fct-cdf head is behind SSDF head, and lies with LSDF head but, starting from the 46th percentile, the fct-cdf of Artemis starts to grow faster than SSDF. This means that 46% of the flows were able to finish faster in SSDF and this is not surprising: SSDF is biased towards flows with a small size/deadline ratio. The 50ms-flows have a ratio that is equal to 5MBps while the 40ms-flows have a ratio that is equal to 8.78MBps. SSDF gains an initial edge in performance by prioritizing the 50ms over the 40ms-flows but, fails short on-the-go when scheduling the 40ms-flows: the 40ms-flows are heavier in size, and starting them will slow-down the 50ms-flows in particular, which leads to a performance degradation. On the other hand, LSDF seems to approximate the flow-scheduling policy that Artemis had learned, but it is still slightly behind Artemis because LSDF is very aggressive towards the 50ms-flows, and if it was not, it would give the flow-scheduler a performance edge on meeting more deadlines. This performance edge will be better highlighted on the next section, where we vary the workload A to be denser with 40ms-flows and this will result in a deadline hit ratio performance drop for LSDF: its aggressiveness towards the 50ms-flows makes the flow-scheduler misses the easy deadlines to meet, and blindly schedule by default the hard deadlines to meet first.

Table 4.6 shows that Artemis was the quickest to finish flows at the 95th percentile, and slightly slower than SSDF but better than EDF on the 96th and the 97th percentiles. Among all four flow-schedulers, Artemis has the shortest and the thinnest tail-latency with the fct-98th, 99th and 100th percentiles being equal to 204.739ms, 1027.525ms and 1029.183ms respectively. Similarly, LSDF has the longest and the heaviest tail-latency

Table 4.6: Head and tail latencies for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently			
Flow Scheduling Policy	FCT 25-th %	FCT 50-th %	FCT 75-th %
EDF	26.395	43.349	75.439
Artemis	27.585	31.132	49.684
SSDF	23.069	38.460	55.955
LSDF	27.573	34.061	50.546
Flow Scheduling Policy	FCT 95-th %	FCT 96-th %	FCT 97-th %
EDF	126.149	133.485	137.649
Artemis	80.411	88.588	94.634
SSDF	86.969	88.037	92.015
LSDF	86.993	1027.525	1029.0123
Flow Scheduling Policy	FCT 98-th %	FCT 99-th %	FCT 100-th %
EDF	175.321	439.984	1039.447
Artemis	204.739	1027.525	1029.183
SSDF	99.328	106.022	1030.071
LSDF	1030.091	4807.770	4847.249

with the fct-98th, 99th and 100th percentiles being equal to 1030.091ms, 4807.770ms and 4847.249ms respectively. The tail latency of EDF and SSDF are almost as short as Artemis but, thicker with SSDF having the thickest one compared to EDF and Artemis. Based on the fct 99th-percentile, and from better to worst, the order is as follows: SSDF, EDF, Artemis, LSDF. Similarly, based on the fct-100th percentile, and from better to worst, the order is as follows: Artemis, SSDF, EDF and LSDF.

Table 4.7: Matching performance metric (x is a match) for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Workload A (variant 1), inter-burst arrival time = 100.15ms, 10 flows served concurrently						
Flow Scheduling Policy	FDH ratio	FSR CDF Head	FSR CDF Tail	FCT CDF Head	FCT CDF Tail	FCT CDF Very-End Tail
EDF		X				X
SSDF		X			X	
LSDF	X		X	X		

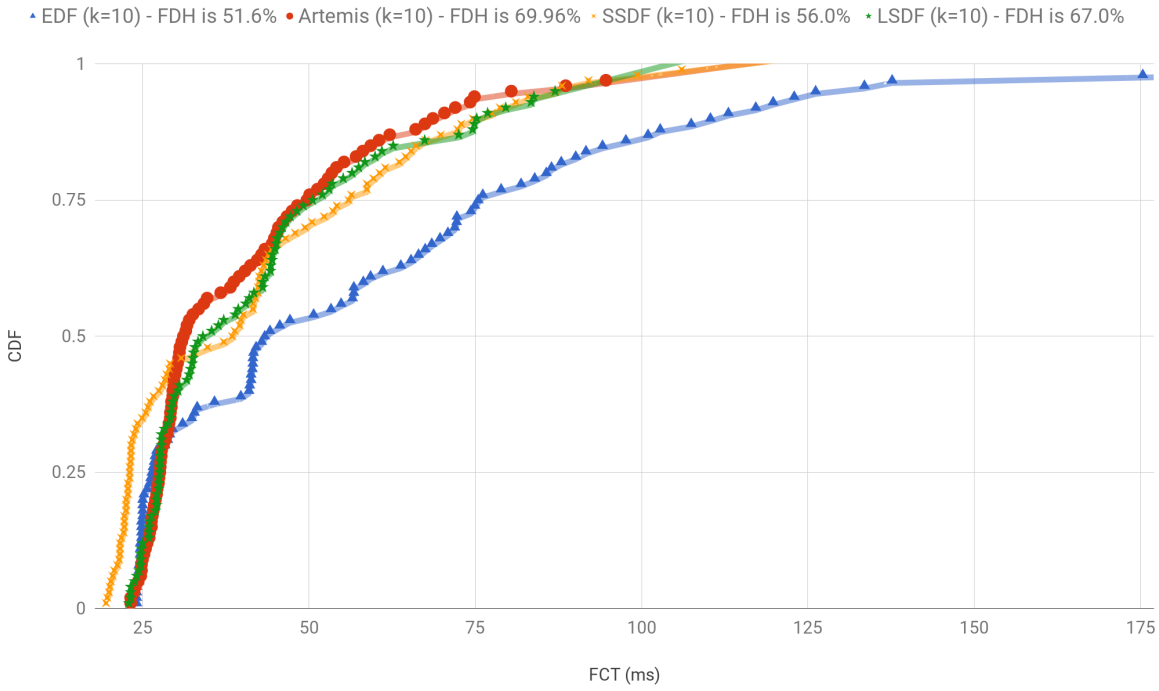

Figure 4.10: Flow completion time cdf for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

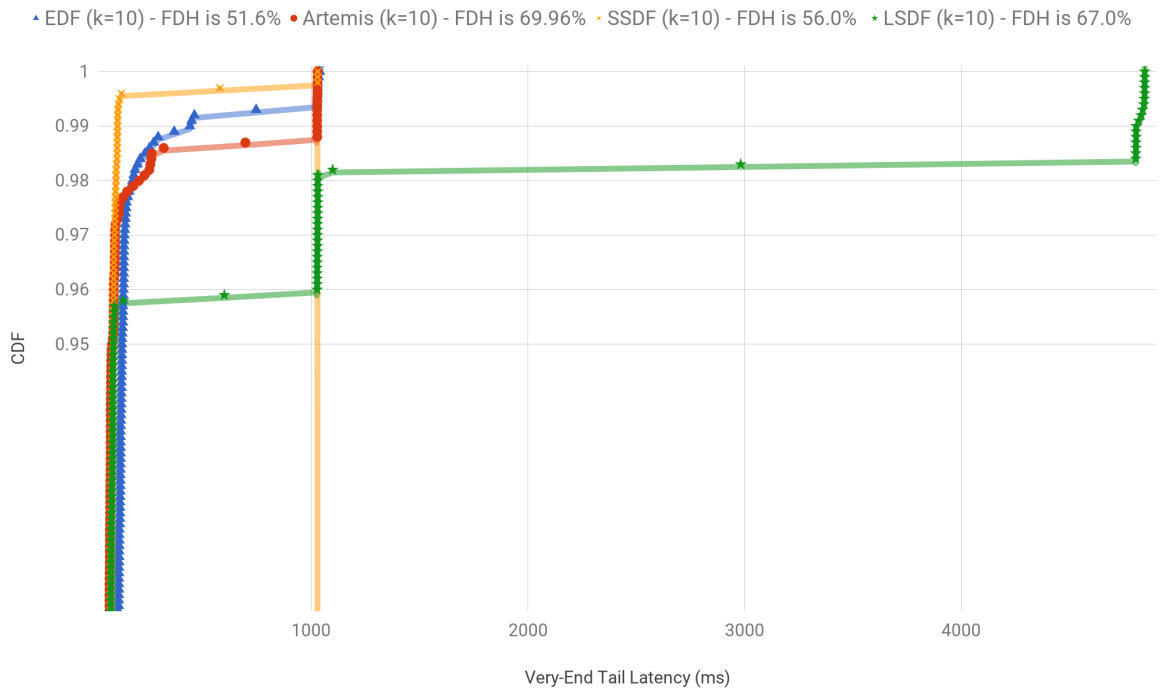

Figure 4.11: Very-end tail latency for Artemis vs EDF, SSDF, LSDF; workload A (variant 1)

Artemis Is Learning a Mixture of SSDF and LSDF

Overall, Artemis was able to meet the maximum number of deadlines for Workload A (variant 1), outperforming EDF with +35.58%, SSDF with +24.93% and performing marginally better than LSDF with +4.42%. Artemis served flows with a rate equals to 98.994 flows/sec that is very close to the experimental flow arrival-time 100.15 flows/sec, minimizing the destination idle time to the least and reducing the waiting time of flows to be served. Artemis has also the shortest and the thinnest tail-latency, and both tail-latencies of EDF and SSDF are almost as short as Artemis but thicker. Nevertheless, Artemis was not the quickest to finish flows in average, and it was SSDF instead. We summarize accordingly all these observations in table 4.7. We therefore expect Artemis to learn a probabilistic flow-scheduling strategy that is composed of a mixture of both strategies: SSDF and LSDF.

4.3 Artemis In Action: Workload A (variant 2)

In the previous section, LSDF and Artemis did almost meet the same number of deadlines. However, we have highlighted the aggressiveness of LSDF against the 50ms-flows having a lower size-deadline ratio compared to the 40ms-flows. In this section, we modify workload A (variant 1) to have more 40ms than 50ms-flows and make the flow-scheduling decisions of LSDF more biased for the 40ms-flows and therefore, aggressive against the 50ms-flows.

The variant 2 of workload A is composed of 60% of (350KB, 40ms) flows and 40% of (250KB, 50ms) flows. Traffic continues to arrive in bursts of 10 flows, where each burst is now composed of 6 flows of (350KB, 40ms) and 4 flows of (250KB, 50ms) as presented in figure 4.12. Compared to variant 1, workload A (variant 2) is denser in 40ms-flows and this makes Earliest Deadline First more aggressive towards the 50ms-flows and in favor of the 40ms-flows. The competition among the 40ms-flows is much higher now and this will cause EDF to meet fewer deadlines than earlier.

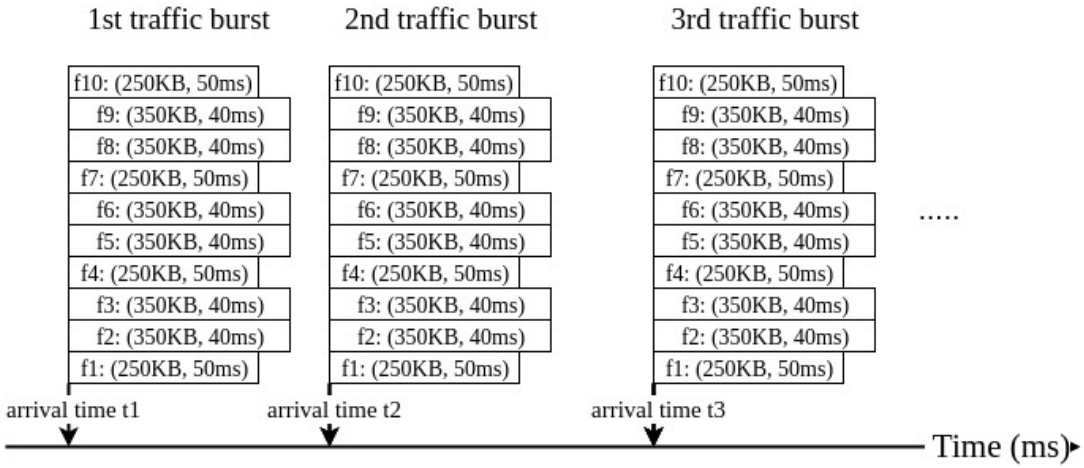

Figure 4.12: Distribution of the gather task workload A (variant 2)

As presented in figure 4.13, LSDF failed short to compete with Artemis this time. Artemis was able to meet 67.94% of the deadlines, accommodating more deadlines than LSDF and SSDF with +16.74% and +16.34% consecutively. both SSDF and LSDF performed comparatively, meeting almost the same volume of deadlines and maxing out around 58%. The deadline hit ratio of EDF was limited to 45.8% which puts Artemis ahead with +48.34% more deadline-met flows. All three flow-schedulers EDF, LSDF, and SSDF, failed to be competitive with Artemis which was able to learn via reinforcement and in real-time

the appropriate flow-scheduling policy to adopt in order to meet the traffic deadlines. In other words, given a collection of arrived flows, Artemis learned a distribution function over this collection of flows to identify which flow to schedule and which flow to defer.

One more time, we do note that performing well on meeting deadlines does not necessarily imply any particular conclusion about the flow completion time metric. Both figures 4.14 and 4.15 show that SSDF has the shortest and the thinnest tail-latency, with the 99th and the 100th percentiles being equals to 95.347ms and 107.509ms regardless that SSDF was meeting less deadlines than Artemis.

In the previous subsection, we claimed that Artemis could be possibly learning a mixture strategy of SSDF and LSDF. This time, none of the three hard-coded flow-schedulers were competitive to Artemis and did not perform comparatively on meeting deadlines. Therefore, we can not make a similar claim in this context, and we can only embrace the power of artificial intelligence for now.

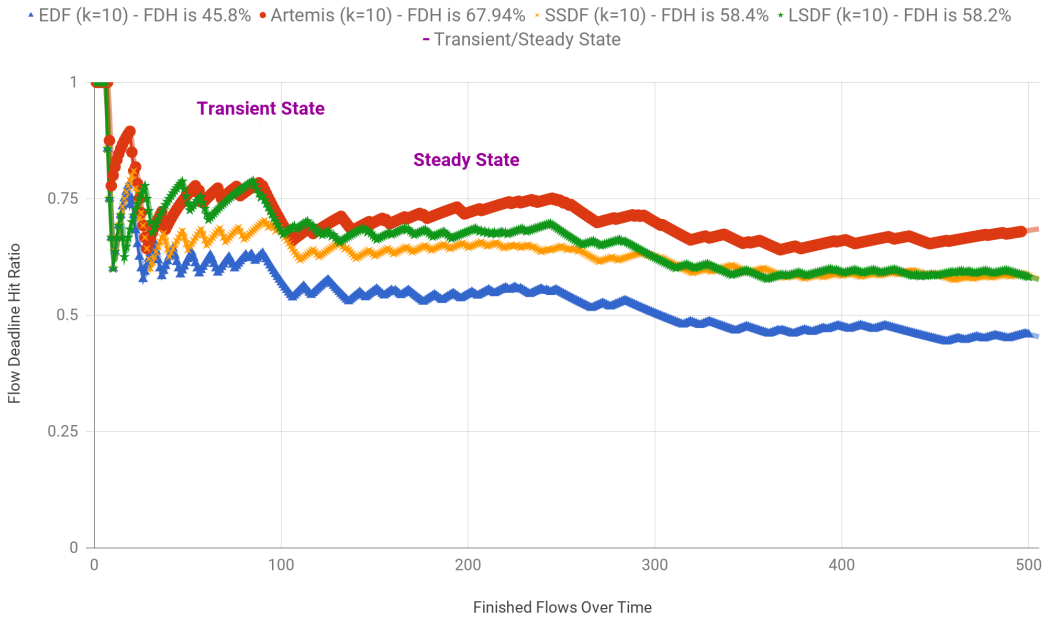

Figure 4.13: Flow deadline hit ratio for Artemis vs EDF, SSDF, LSDF; workload A (variant 2)

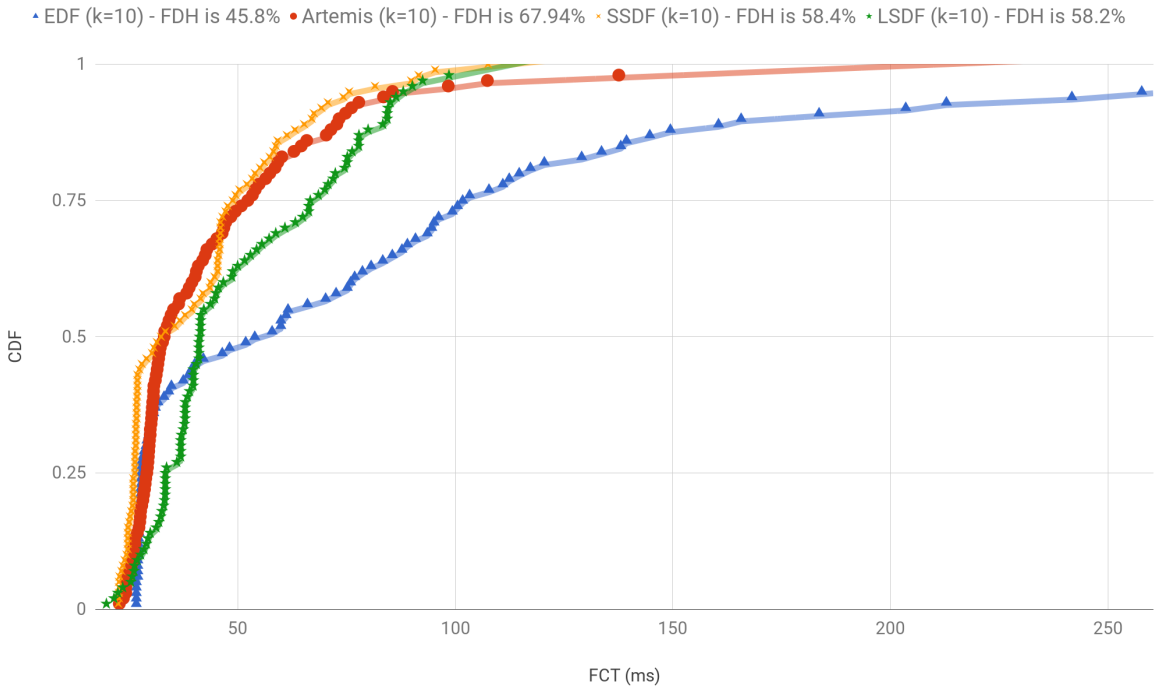

Figure 4.14: Flow completion time cdf for Artemis vs EDF, SSDF, LSDF; workload A (variant 2)

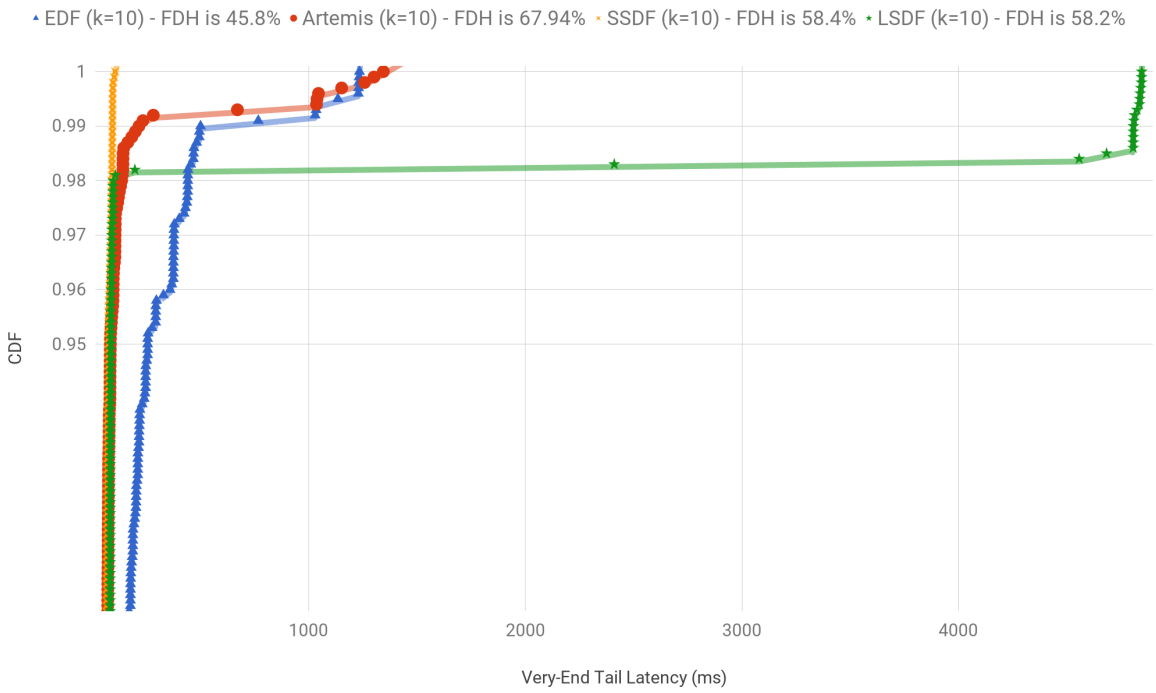

Figure 4.15: Very-end tail latency for Artemis vs EDF, SSDF, LSDF; workload A (variant 2)

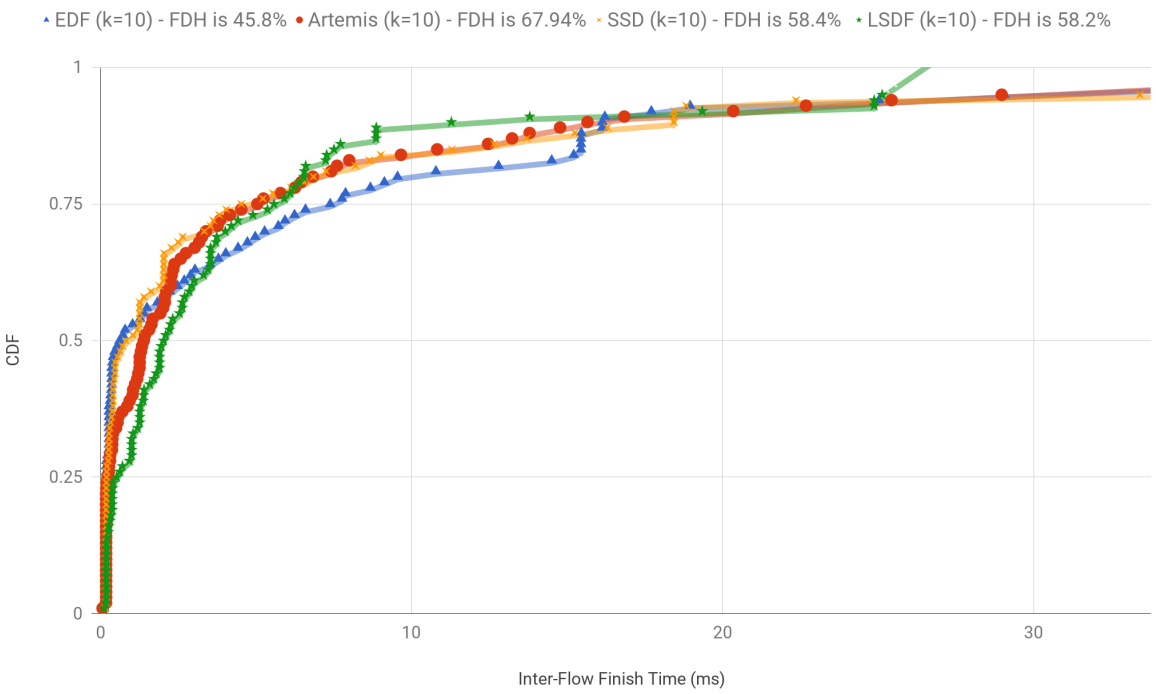

Figure 4.16: Inter-flow finish time cdf for Artemis vs EDF, SSDF, LSDF; workload A (variant 2)

Chapter 5

Conclusion and Future Work

Serving online users' requests in a datacenter environment relies on meeting the deadlines of the generated scatter/gather flows per each request, to return a response in a timely fashion. Rather than modeling the characteristics of workload's traffic flows, and counting on the human intuition to come up with a flow-scheduling strategy, we proposed in this thesis to build a flow-scheduler that is driven by the requirements of the workload and learns how to schedule the generated traffic using a reinforcement-learning loop.

We presented Artemis: A token-based workload-driven flow-scheduler at the end-hosts. In Artemis, we allow a source to start sending a particular flow upon requesting and acquiring a token from the destination side. The token-request is issued by a source to the particular destination of the flow, and it does expose the requirements of the flow. At the destination side, we modelled the flow-scheduling problem as a deep reinforcement learning problem, and we solved it using the actor-critic architecture.

We used two specific gather workloads (1) to demonstrate the ability of Artemis to learn how to schedule deadline flows on its own and (2) to evaluate its effectiveness to meet deadlines. We compared Artemis against the universal deadline-aware flow-scheduler Earliest Deadline First (EDF) and two additional rule-based flow-schedulers that are aware of both the deadlines and the sizes of the flows: Largest Size Deadline ratio First (LSDF) and Smallest Size Deadline ratio First (SSDF). We showed that Artemis is able to learn how to schedule deadline flows and adjust the flow-scheduling strategy it is learning with the traffic-requirements.

In this work, we had evaluated Artemis using two specific gather workloads that are being collected at one static destination. The next step would be to evaluate Artemis using gather workloads that are being collected at different locations. In addition to gather

traffic, scatter traffic is quite common in modern datacenters, and this would be the next fundamental workload to test Artemis with.

References

[1] *Google deep learning framework tensorflow*. https://www.tensorflow.org/.

[2] *The ns-3 discrete-event network simulator*. https://www.nsnam.org/.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 19–19, 2010.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM computer communication review*, volume 40, pages 63–74. ACM, 2010.

[7] Mohammad Alizadeh, Shuang Yang, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Deconstructing datacenter packet transport. In *Proceedings of the 11th ACM Workshop on hot topics in networks*, pages 133–138. ACM, 2012.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[9] Alexey Andreyev. *Introducing data center fabric, the next-generation Facebook data center network*, 2018 (accessed May 28, 2018). https://code.facebook.com/posts/360346274145943/.

[10] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 25. ACM, 2014.

[11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[12] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[13] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 65–72. ACM, 2009.

[14] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[15] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of the 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564. IEEE Publ. Piscataway, NJ, 1995.

[16] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187. ACM, 2016.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Facebook.com. *Request access to the packet trace of Facebook Altoona datacenter*, 2018 (accessed May 28, 2018). https://www.facebook.com/network-analytics.

[19] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 1. ACM, 2015.

[20] Alex Hern. *Google's Go-playing AI still undefeated with victory over world number one*, 2017 (accessed May 28, 2018). https://www.theguardian.com/global/2017/mar/14/googles-deepmind-makes-ai-program-that-can-learn-like-a-human.

[21] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 127–138. ACM, 2012.

[22] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[23] Virajith Jalaparti. *Speeding up Distributed Request-Response Workflows*, 2013 (accessed April 23, 2018). https://conferences.sigcomm.org/sigcomm/2013/slides/sigcomm/20.pptx.

[24] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 219–230. ACM, 2013.

[25] R. Krzanowski. *Burst (of packets) and Burstiness*, 2006 (accessed April 23, 2018). https://www.ietf.org/proceedings/66/slides/ippm-10.pdf.

[26] R. Krzanowski. *Data Center TCP (DCTCP): TCP Congestion Control for Data Centers*, 2017 (accessed May 27, 2018). https://tools.ietf.org/html/rfc8257.

[27] Changhyun Lee, Chunjong Park, Keon Jang, Sue B Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *USENIX Annual Technical Conference*, pages 403–415, 2015.

[28] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 319–330. ACM, 2011.

[29] Ishai Menache, Shie Mannor, and Nahum Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238, 2005.

[30] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *Proceedings of the 14th ACM workshop on hot topics in networks*, page 24. ACM, 2015.

[31] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.

[32] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[33] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. *ACM SIGCOMM Computer Communication Review*, 44(4):491–502, 2015.

[34] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[35] Kadangode Ramakrishnan, Sally Floyd, and David Black. The addition of explicit congestion notification (ecn) to ip. Technical report, 2001.

[36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[37] Ian Sample. *Googles DeepMind makes AI program that can learn like a human*, 2017 (accessed May 28, 2018). https://www.theguardian.com/global/2017/mar/14/googles-deepmind-makes-ai-program-that-can-learn-like-a-human.

[38] Ian Sample. *'It's able to create knowledge itself': Google unveils AI that learns on its own*, 2017 (accessed May 28, 2018). https://www.theguardian.com/technology/2017/may/25/alphago-google-ai-victory-world-go-number-one-china-ke-jie.

[39] Ian Sample. *Google DeepMind's AI program learns human navigation skills*, 2018 (accessed May 28, 2018). https://www.theguardian.com/technology/2018/may/09/googles-ai-program-deepmind-learns-human-navigation-skills.

[40] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[41] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[42] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM computer communication review*, volume 45, pages 183–197. ACM, 2015.

[43] Statista.com. *Data center workloads, by installed application, worldwide from 2015 to 2020 (in millions)*, 2018 (accessed May 28, 2018). https://www.statista.com/statistics/638056/worldwide-data-center-workloads-by-application/.

[44] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

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

[46] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI*, pages 407–420, 2017.

[47] Wikipedia. *Burstiness*, 2013 (accessed April 23, 2018). https://en.wikipedia.org/wiki/Burstiness.

[48] Wikipedia.org. *Pareto principle*, 2018 (accessed May 28, 2018). https://en.wikipedia.org/wiki/Pareto_principle.

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

Glossary

k The max number of concurrent flows to start by the flow-scheduler. 23

D2TCP Data-Aware Datacenter TCP. 1, 9

D3 Deadline-Driven Delivery Control Protocol. 1

DCTCP Data Center Transmission Control Protocol. 4, 9

ECN Explicit Congestion Notification. 2

EDF Earliest Deadline First. 2, 3, 21, 24, 33

FCT Flow Completion Time. 23

FDH Flow Deadline Hit ratio. 24, 37

FIFO First In First Out. 23

FSR Flow Service Rate. 37

IETF Internet Engineering Task Force. 4

LSDF Largest Size Deadline ratio First. 3, 21, 33–35, 38, 39

NIC Network Interface Controller. 5

OLDI Online Data-Intensive applications. 6

PDF Probability Distribution Function. 33

RTT Round Trip Time. 5

SSDF Smallest Size Deadline ratio First. 3, 21, 33, 34, 38–40

SSF Shortest Size First. 9

TCP Transmission Control Protocol. 4, 5

UDP User Datagram Protocol. 4, 5