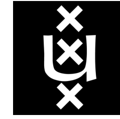
Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master Thesis

A Learning-based Approach for Stream Scheduling in Multipath-QUIC

Author: Marios Evangelos Kanakis (2619105)

1st supervisor: Lin Wang
2nd reader: Henri Bal

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

September 24, 2020

"We are what we repeatedly do. Excellence, then, is not an act, but a habit."
from The Story of Philosophy*, by William James Durant*

Abstract

Proliferation of networked devices with multiple network interface capabilities stipulate progress in transport protocols. Recent advancements integrate multi-path features towards bandwidth aggregation benefits and increased reliability by seamless handovers. A distinct effort is Multipath-QUIC (MPQUIC), a successor of QUICK UDP Internet Connections (QUIC). QUIC is the recently proposed high-performance transport protocol that lies on the application-layer and relies on UDP to operate. QUIC offers reduced latency in connection instantiation and is secure by default. MPQUIC provides support for devices with multiple interface links to boost performance gains. On the other hand, loading time of web pages, a crucial factor for user quality of experience (QoE), is hardly reduced. Recent works indicate scheduling policies on MPQUIC are critical for performance gains and QoE for web browsing, whereas suboptimal scheduling decisions result in degradation.

In this thesis, we propose SAILfish, a novel learning-based stream scheduling system for MPQUIC. We utilize state-of-the-art Deep Reinforcement Learning techniques to generate an efficient scheduling policy. With a strong focus on QoE, we design and set the objective for SAILfish towards reduction of stream completion times. By comparing vanilla stream scheduling mechanisms of MPQUIC under different network settings and various websites, we argue that a learning-based scheduler design is a viable alternative to existing heuristic-based solutions. Results indicate SAILfish reduces average completion time of streams in a variety of configurations compared to MinRTT stream-based vanilla scheduler. Especially, in very-low and high bandwidth configurations, SAILfish manages an average 22% and 11% reduction of loading times across multitude and variant websites, respectively. On the other hand, we exhibit that a comparison between a stream and a packet-based mechanism is not efficient yet, due to the vital differences of the two schemes.

Acknowledgements

I would like to express my deep gratitude to Dr. Lin Wang, my research supervisor, for his valuable guidance for this work. His ardent encouragement, constructive criticism, and insightfulness have been greatly appreciated. Without him, this thesis would have not been the same.

I would also like to extend my thanks to Prof. Henri Bal, for his willingness to be the second reader for this thesis.

Finally, I would like to acknowledge the support of my family and friends throughout the course of this research. Especially, I wish to thank Miss Rodanthi Kyriakaki for her love and understanding during the challenges of this work.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background 7

2.1 QUIC: A Transport Protocol for the Internet 7

2.1.1 The Goals of QUIC 8

2.1.2 The Design of QUIC Protocols 9

2.1.3 State of the Art 10

2.2 Bringing Multiple-Paths to QUIC 10

2.2.1 The Design of MPQUIC 10

2.2.2 QUIC vs. MPQUIC 12

2.3 Reinforcement Learning 12

2.3.1 Model-Free Policy Gradient RL 13

2.3.2 Neural Networks as Function Approximators 14

2.3.3 The Case for Advantage Actor-Critic 14

3 Related Work 17

3.1 Scheduling in MPTCP 17

3.2 The Case for MPQUIC 18

4 SAILfish: Scheduling Agent In Learning of Multipath-QUIC 21

4.1 System Overview 21

4.1.1 Learning-based Scheduling 22

4.1.2 Traffic Monitoring 23

4.1.3 Limitations of the Design 24

4.2 Learning Scheduling Policies 24

4.2.1 Problem Formulation 24
4.2.1.1 State Space 25
4.2.1.2 Action Space 26
4.2.1.3 Reward 26
4.2.2 Learning Algorithm 27
4.2.2.1 A2C-based Learning 27
4.2.2.2 A2C vs. A3C 27

5 Evaluation 29
5.1 Implementation 29
5.1.1 Setting up MPQUIC 29
5.1.2 SAILfish in Detail 30
5.1.3 Module Communication 32
5.1.4 Training SAILfish 32
5.2 Experimental Setup 33
5.2.1 Dependency Graphs 33
5.2.2 Network Configurations 34
5.2.3 Baselines 34
5.2.4 Evaluation Environment 35
5.3 Performance Evaluation 36
5.3.1 Collecting Scheduling Data 36
5.3.2 Performance Analysis 36
5.3.2.1 SAILfish vs. Stream-based MinRTT 36
5.3.2.2 SAILfish vs. Per-packet MinRTT 38

6 Discussion 43
6.1 Stream-based vs. Per-Packet Scheduling in MPQUIC 43
6.2 SAILfish vs. Stream-based MinRTT 44
6.3 Threats to Validity 45

7 Limitations 47
7.1 The Why Behind The Action 47
7.2 Unexplored Territories 48
7.3 Computational Adversities 48
7.4 Parameter Selection 48

8 Conclusion 51

References 53

List of Figures

2.1 A high-level overview of a RL environment formulated as an MDP. 13

4.1 High-level overview of SAILfish. 22
4.2 Typical heuristics stream-based scheduler in MPQUIC. 24
4.3 High-level overview of Actor-Critic algorithms. 28

5.1 Training environment of SAILfish. 31
5.2 Performance of SAILfish on small dependency graphs under different network configurations. 37
5.3 Performance of SAILfish on medium dependency graphs under different network configurations. 39
5.4 Performance of SAILfish on large dependency graphs under different network configurations. 41

List of Tables

5.1 Dependency graphs used in evaluation. 33
5.2 Multipath configurations for network simulation. 35

LIST OF TABLES

1

Introduction

Global trends in network traffic and booming mobile device connectivity [1], set the pace for advancements in networking protocols. Efforts to diminish performance bottlenecks [2] and potential degradation of networks in legacy stack, e.g., Bufferbloat [3] and Incast [4], have emerged several aspiring works [5, 6, 7]. Most of the approaches, focus on the layer that suffers the most, the transport layer.

QUIC [5] is one of the recent works that exhibit significant performance benefits [8] over traditional HTTP(S) stack (e.g., TCP, TLS, etc.). In fact, QUIC is widely *spreading* within Google's datacenters [8], and support in popular browsers has initiated officially (e.g., Chrome, Firefox, etc.) [9]. QUIC writes the future by multiplexing streams and sessions over UDP, without the extra hassle to intervene at kernel-level, as it lies on the application layer.

Meanwhile, Multipath TCP (MPTCP) [10], a recently proposed backwards compatible augmentation for TCP, provides multipath support. Further, MPTCP leverages subflows to integrate multiple standard connections between two endpoints, i.e., a client and a server, where each subflow represents a distinct path. To ensure reliability in data transportation, MPTCP introduces sequence numbers for in-order delivery, by proposing Data Sequence Signal (DSS) options [10]. Packet delivery between subflows is dictated by a proposed scheduling mechanism. The default scheduling policy in Linux kernel implementation is MinRTT [11]. MinRTT relies on heuristics to operate, and in particular, selects the subflow with lowest latency to transmit a packet. Overall, MPTCP highlights substantial performance boosts by utilization of multipath resources [12, 13, 14].

Despite success, QUIC does not provide built-in multiple path capabilities. This lack in support is on the contrary with proliferation in cellular connectivity [1, 15] and increasing amount of dual-stack hosts (i.e., IPv4 and IPv6) [16]. It is only recently, that Coninck et

al. integrating on top of QUIC, enable utilization of several paths [16] and have submitted for standardization by the IETF [5]. Similarly to MPTCP, Coninck et al. propose Multipath QUIC (MPQUIC) to enable bandwidth aggregation, fault-tolerance (i.e., seamless handovers in case of path failure), and overall, enhanced user experience [16].

However, by design, multiple path protocols are subject to additional challenges compared to their single path counterparts. Perhaps the most arduous task of all the challenges, is the scheduling of packets. It has been shown that suboptimal decisions may result in degraded performance [17]. The default scheduler of MPTCP, i.e., MinRTT [11], which MPQUIC integrates [16], is incapable of providing sufficient scheduling decisions, especially when paths are heterogeneous [18] (e.g., discrepancy in bandwidth or latency between paths), or on imminent stream prioritization schemes [15] (e.g., scheduler does not consider stream priorities as in HTTP/2).

Therefore, scheduling in MPQUIC has attracted a lot of attention recently and several novel solutions have been proposed [15, 18, 19, 20]. FStream [19] offers a stream-based scheduling approach dependent on file size and priorities. Subsequent work of Shi et al. propose PriorityBucket [20] that targets for first rendering time. SA-ECF [18] performs per-packet scheduling on completion time estimates, whilst, other attempts, assume apriori knowledge of the environment [15].

All these approaches have one thing in common; they rely on heuristics to operate. Heuristic-based schedulers optimize for a specific objective (e.g., bandwidth aggregation) or perform under a certain criterion (e.g., assigned priorities), metric (e.g., latency) or assumption (e.g., symmetrical paths). It is a common pitfall for such attempts to exclusively target their goal and come out rigid, i.e., incapable of adjusting under new circumstances [17].

On the other hand, we have witnessed tremendous growth in Machine Learning (ML) far-reaching applications. In particular, ML has been successfully applied to a plethora of challenges [21, 22, 23]. Besides the common domains of prevalence of ML, recently, we observe efforts to leverage learnt-based solutions in networking systems [24, 25, 26, 27, 28].

Taking into consideration the challenge of scheduling packets in multipath protocols, and following the recent progress in ML, we formulate our research questions, accordingly. *Can we learn an efficient stream scheduling policy for MPQUIC?*. Moreover, we attempt to answer the following questions: (1) *How can we formulate scheduling in MPQUIC as a reinforcement learning task?*, (2) *What are the implications of learning-based approaches for scheduling?*, and (3) *How can we effectively incorporate RL into a transport protocol?*.

Some early attempts have already been made to apply Deep Reinforcement Learning (DRL) in multipath scheduling. ReLeS [17] leverages modern DRL techniques and algorithms to derive a scheduling policy for MPTCP. ReLeS combines historic and current networking observations to make informed packet scheduling decisions. The optimization objective is a factor of Quality of Service (QoS) attributes [17]. Results indicate superiority amongst state-of-the-art baselines while the system is capable of adapting to unforeseen networking conditions [17]. Wu et al. propound a hybrid system to combat scheduling deficiencies in state-of-the-art schemes concerning MPQUIC [29]. Peekaboo leverages contextual Multi-Armed Bandits (MAB) [30] with heuristic-based policies and stochasticity adjustment strategy [29] to deliver a robust and flexible packet-based mechanism. However, Peekaboo does not consider prioritized streams, and multiple concurrent streams in general. Instead, focus of evaluation is given on single-file downloading and basic streaming scenarios.

Drawing inspiration from latest advances, in this thesis, we propose Scheduling Agent in Learning of MPQUIC (SAILfish). SAILfish is a novel learning-based stream scheduling system. To the best of our knowledge, SAILfish is a first-step in deriving a robust, self-adjusting system that concerns stream-based scheduling decisions in Multipath-QUIC. SAILfish is a networking system that learns only by pure exploration of the environment, and is able to generate scheduling policies on HTTP/2 prioritized streams.

Similarly to ReLeS, SAILfish adopts a state-of-the-art DRL algorithm to derive its policy. Training is conducted with synthetic network traces and website dependency graphs in an emulation of client-server MPQUIC architecture. SAILfish considers several crucial factors that constitute for its reinforcement signal (i.e., reward). In general, SAILfish favors fast paths (i.e., high-throughput, low-latency, etc.) and punishes for slow stream completion times, and lossy paths. This allows for a dynamic, efficient scheme that takes into consideration vital metrics of networked systems, whilst, maintaining a user cedric objective, i.e., stream completion times and website content download. In addition, SAILfish is a fully-fledged system capable of *self-healing* through online policy updates, when networking conditions fluctuate.

Comparable to most scheduling mechanisms for MPQUIC that concern stream-to-path allocations, SAILfish is subject to certain inherent limitations. Particularly, SAILfish responds to bursts of scheduling requests and outputs the optimal path for transmitting a stream. In this manner, SAILfish is not able to estimate potential packet blockage, since it is not aware apriori, i.e., at the time of serving the requests, if congestion on the selected paths will be induced. Therefore, SAILfish does not proactively prevent congestion,

in contrast to packet-based approaches that are able to make ad-hoc decisions to prevent path overflow.

Results and contributions The key results and contributions of this thesis are summarized as follows.

- We propose a novel learning-based approach in generating scheduling policies for MPQUIC. To the best of our knowledge, SAILfish is the first attempt in deriving a learnt stream-based scheduling policy targeting performance challenges in multipath protocols with respect to user QoE.
- We formulate stream-based scheduling in multiple path environments as a reinforcement learning task. Further, we design and elaborate on the reward function to optimize MPQUIC scheduling decisions under various websites and network configurations. Moreover, we propose a state-of-the-art deep reinforcement learning algorithm for training SAILfish by pure exploration of the environment.
- We contribute a thorough system design, where SAILfish is incorporated into a real-word deployment scenario, and fundamental software components are presented and specified.
- We provide tangible evidence SAILfish is performing better than stream-based baseline. In particular, we implement and evaluate SAILfish under emulated network configurations and website dependency graphs. We design and conduct experiments based on user QoE, and exhibit SAILfish outperforms vanilla stream-based MinRTT in average loading time of web pages.

Outline The remainder of this thesis is organized as follows. In Chapter 2, we provide a comprehensive background on QUIC, Multipath QUIC, and Reinforcement Learning. Chapter 3 presents related work regarding scheduling schemes for multipath protocols, and specifically, efforts in MPTCP and MPQUIC. Following related work, we dedicate a chapter, Chapter 4, to SAILfish. In this chapter, we discuss the specifics of our system design and introduce SAILfish components. Additionally, we formulate the stream scheduling problem as a RL task. We elaborate on the state, action, and reward design. Chapter 5 regards the evaluation of our proposed system. Initially, we discuss the implementation in detail, and proceed to argue experimental design, followed by an extensive performance evaluation of SAILfish. We discuss the results of experimentation in Chapter 6, and argue

performance benefits of SAILfish over traditional solutions. Further, we proceed to introduce certain threats to validity that affect our research. Then, in Chapter 7 we present limitations of our approach. Finally, in Chapter 8 we conclude our research and summarize our contributions.

2

Background

In this chapter, we provide readers with comprehensive background knowledge on the three basic pillars of interest that ultimately compose SAILfish. These are, (1) QUIC, (2) MPQUIC, and (3) RL. First and most importantly, we introduce QUIC protocol, argue its goals, explore the design and elaborate on fundamental theoretical concepts. We proceed to specify our case by augmenting the concept of QUIC into that of multiple paths, providing necessary information. Further, a brief comparison between the two protocol specifications is presented to argue the benefits of multipath capabilities. The background section is concluded with an essential introduction on RL fundamentals. Specifically, we present the case for RL and attempt to familiarize readers with basic concepts that SAILfish utilizes to achieve its goals.

2.1 QUIC: A Transport Protocol for the Internet

QUIC, initially introduced as Quick UDP Internet Connections [31] and later drafted as plain *QUIC* (i.e., not an acronym) by IETF [5] is a transport-layer protocol that relies on UDP. Initially proposed by Google to reduce loading times of web pages, QUIC now attempts to substitute the TCP, TLS, and HTTP stack altogether bringing overall improved performance and low-latency client-server communication. Google has already deployed QUIC since 2012 and as of 2015 about half of Chrome's browser requests are served through it [8]. More specifically, Google has recorded a 3% reduction in average page loading time as well as a 30% reduction in rebuffering events on YouTube as reported by users [8]. Albeit the results in search experience might seem insignificant, it must be taken into account that Google's search supports *pre-established connections* whilst the site itself is heavily

optimized towards serving results as fast as possible [8]. Hence, it is safe to assume that QUIC has been quite successful in delivering its promises.

2.1.1 The Goals of QUIC

As Internet traffic sharply increases annually [1], most of today's websites and services still rely on HTTP and the underlying protocols (i.e., TCP and TLS), to serve content to users. That being said, the initial vision and design of QUIC is heavily inspired by the inadequacies of underlying infrastructure and existing limitations.

First of all, TCP's design is based on network observations of its time [2]. As a result, TCP lacks in flexibility and is hampering efforts to scale in accordance to modern Internet traffic requirements [2]. A notorious performance bottleneck of TCP is head-of-line (HOL) blocking [31]. Head-of-line blocking occurs in a TCP stream, when a single packet is lost, and subsequent packets wait until it is (re-)transmitted and received. HTTP/2, a recent revision of the HTTP protocol, puts effort into addressing HOL blocking by multiplexing requests over the same TCP connection, yet, this approach is only effective on the application-layer. Since, HTTP/2 relies on TCP to operate, HOL blocking still persists on packet-layer. Another weak point of TCP, is inability to handle network congestion adequately, which often results in degraded performance [31]. Despite several attempts [32, 33, 34], end-to-end congestion control schemes tune the congestion window under a certain policy, typically based on heuristics. Most of the approaches are drafted on unrealistic assumptions about the network, especially in cases of emergent and ad-hoc topologies, leading to policies that underperform in the wild. Therefore, TCP's congestion control mechanisms are often associated with reduced performance.

On the other hand, TLS comes with its own deficiencies. Two notorious disadvantages of TLS as described in [31] are a) induced RTT overhead before initial data transportation, and b) redundant packet size. The former regards an implementation detail rather than a security trait. In detail, TLS takes place before data is transmitted either way, and is responsible for secure communication between endpoints. As part of the TLS implementation, a three-way handshake must initiate before exchanging any *meaningful* information, where client and server negotiate the version and exchange cryptographic keys. This induces at least an additional RTT delay in communication [31]. Regarding redundancy in packet size, earlier versions of TLS, had a decryption in-order dependency, i.e., earlier packets had to be decrypted first. To meditate this inherit flaw, since TLS 1.1, initialization vectors have been introduced, which constitute for bigger packets in size [31].

2.1.2 The Design of QUIC Protocols

In order to replace the existing stack without intervening at low level protocols and dictating significant alterations to existing systems, QUIC is designed on the premises of existing infrastructure. First, QUIC employs UDP to tackle the interoperability challenge. This diminishes the need to modify the underlying transportation protocols of current systems. Second, to further ease its integration, QUIC operates in the application space rather than kernel space. Doing so, developers of applications can directly shift to QUIC to serve content. Both design decisions, ensure that QUIC is able to run everywhere without significant modifications (passing ossification from middleboxes, user space access, etc.) [31].

QUIC avoids HOL blocking by leveraging the concept of streams. In particular, QUIC abstracts raw data transferring with streams. Each stream represents a unidirectional or bidirectional data flow (i.e., channel) from client-to-server and vise versa [5]. Moreover, streams are uniquely identified, and their ID's are distinguished based on the initiator, even numbers for clients and odds for servers. On a lower level, streams are composed of STREAM frames that instantiate a stream and carry stream data [5]. Whenever a QUIC packet is lost or delayed, only streams inside this particular packet are blocked. Hence, HOL blocking witnessed in TCP is largely avoided.

Moreover, QUIC is designed to be multiplexed out-of-the-box. QUIC connections are composed of several streams that can be initiated from both endpoints. As stated already, a stream consists of one or more STREAM frames. To transmit streams, QUIC bundles frames into packets. Therefore, QUIC packets are composed of STREAM frames. By bundling one or more frames into a single packet, QUIC achieves multiplexing [5].

On the other hand, QUIC is secure by default. Initially, QUIC connection establishment involves a secure handshake [16], similar to TLS. Further, QUIC encrypts packet payloads from the start, ensuring connections are always encrypted, and effectively avoiding middlebox ossifications. A byproduct of this inherit secure design, is the fact that QUIC reduces the amount of necessary RTT's to one, enabling faster establishment of connections, and eventually, faster transfer of *meaningful* data.

Finally, version upgrades in QUIC are easily deployable. In the secure handshake, necessary for connection establishment between two endpoints, clients and servers are able to negotiate versions, with preference for highest supported version. Thus, upgrade of QUIC versions in existing systems and infrastructure is seamless.

2.1.3 State of the Art

Early evaluations of QUIC exhibit boosts in performance and essentially in latency and loading times of web pages [8, 35, 36, 37]. More specifically, Das reports that QUIC outperforms HTTP/1.1 on very-low and low bandwidth links, and continues improving as RTT increases [35]. Preliminary positive results in [8] are further backed by Google in [36] where Langley et al. indicate reduction in search and video streaming latency. Biswal et al. report better overall performance in poor networking conditions [37].

Despite the good results in low-bandwidth, high-latency, and high-loss environments, [36] and [37] agree that on overall good networking conditions, QUIC is rarely an improvement and sometimes under-performs in comparison to HTTP. In addition, Megyesi et al. in a comprehensive comparison between QUIC, SPDY [6], and HTTP report similar performance with an emphasis on web page loading times.

2.2 Bringing Multiple-Paths to QUIC

Although QUIC is a modern transport protocol, it does not address multiple paths out-of-the-box. As a path, we define a connection between a client and a server. For instance, mobile devices often have several links to wireless connectivity (e.g., LTE, WiFi, etc.). Multipath-TCP (MPTCP), which supports multipath connections has already been proposed for standardization [10]. MPTCP identifies subflows, i.e., paths, during connection establishment and introduces DSS options to enable in-order data transportation from multiple paths [10]. Besides newly introduced TCP options, at network layer, data segments look identical to standard TCP flows [10]. In this manner, MPTCP does not necessitate changes in applications to operate.

Multipath transport brings new opportunities for improving network throughput and reliability. Deng et al. observe better performance with MPTCP in applications with long flows [12]. Furthermore, Raiciu et al. report increased throughput through higher link exploration in large-scale data-centers [13]. In [14], Paasch et al. conduct an experimental study in link handovers and result in a *smooth handover experience* with Full-MPTCP mode.

2.2.1 The Design of MPQUIC

Following the recent advancements in MPTCP, Coninck et al. propose and submit a draft for standardization, introducing Multipath-QUIC (MPQUIC) [16, 38]. MPQUIC is an ef-

fort to leverage and establish multiple path utilization in the QUIC protocol. Concerning design and implementation of MPQUIC, the authors argue the need for selecting optimal paths when network conditions differ (e.g., IPv6 over IPv4), and fault-tolerance when a path fails by handing over the connection to another link [16]. MPQUIC comes into realization by tackling five challenges. Namely, Coninck et al. augment QUIC with modifications on: a) Path Identification, b) Reliable Data Transmission, c) Path Management, d) Packet Scheduling, and e) Congestion Control [16].

Path identification solves ambiguity where packets with lower packet number from one path might arrive after packets with higher packet number from another path have been received. In such a case, middleboxes might drop a packet with a lower packet number [16]. In addition, uniquely identifying paths enables multiple flows in change of address.

Besides identification of paths and to ensure reliability in data transmission, MPQUIC uses a separate packet number space per path. Doing so, the authors combine path IDs and packet numbers in the public header, achieving a potential reduction in huge ACK frames. Also, it enables acknowledgments of packets from different paths by the receiver [16].

Both endpoints are able to create paths and thus dictates the need for path management. MPQUIC offers a path manager that supports path creation and deletion. ID collision of paths is avoided by following the same pattern as in stream IDs[1]. Hence, clients use odd numbers, and servers even respectively. An ADD_ADDRESS frame is added to advertise potential paths over dual-stack hosts.

Packet scheduling on MPQUIC relies on the same heuristic as the default MPTCP implementation on Linux Kernel [11, 16]. Packets are scheduled on an iterative manner on the path with the lowest smoothed-RTT measurements whilst being constrained by the congestion window (i.e., path's congestion window is not full). MPQUIC offers more flexibility than MPTCP, as each packet can be transmitted over any available path, meanwhile MPTCP is restricted to transmit in sequence to avoid ossification from middleboxes [16]. Current implementation of MPQUIC favors *all-paths* utilization by initially duplicating traffic over available paths when networking statistics related to a new path are yet unknown [16].

In order to reduce unfairness, Coninck et al. leverage OLIA scheme for congestion control, as results indicate better performance than CUBIC [32] with MPTCP [16]. OLIA is pareto-optimal and therefore can avoid throughput utilization problems of TCP and MPTCP users [39].

[1] Current implementation of MPQUIC does not support server-initiated paths.

2.2.2 QUIC vs. MPQUIC

Overall, results in performance evaluation of MPQUIC validate the ones presented in Section 2.1.3. Therefore, in low-bandwidth and high-loss environments, MPQUIC performs better than MPTCP and is advantageous to QUIC. On the other hand, when the networking conditions are better, MPQUIC performs slightly better than MPTCP in most scenarios [16]. However, Coninck et al. report that there are no gains from utilizing multiple paths when considering short-file transfers, where initial connectivity is the major contributing factor to performance [16]. Regarding network handover, MPQUIC is able to manage failing or unstable paths by adding an extra PATHS frame in retransmitted packets [16].

2.3 Reinforcement Learning

Reinforcement learning (RL) is essentially a decision making process. Therefore, the keyword learning is used in the context of learning to map stimuli from the environment to decisions, or better known as actions [40]. Unlike supervised and unsupervised learning, RL has by definition an exploratory nature. The agent (i.e., learner) observes the environment, interacts with it, and obtains a reward based on the impact. As an objective, the agent wants to receive the maximum cumulative reward.

To obtain a high reward the agent must strike a balance between exploration and exploitation [40]. It is impossible for an agent to be effective (i.e., take actions that yield high reward) without exploiting, but exploitation implies that the agent has previously explored the environment and derived to optimal mappings between states and actions. Hence, exploration/exploitation is a common challenge in RL.

Formally, RL problems can be mathematically modelled as Markov Decision Processes (MDPs) [41]. MDPs frame decision making problems that have a sequence naturally, i.e., actions taken now, affect the environment in the "future" [40]. In Figure 2.1 a basic agent-environment interaction is depicted. The agent observes the state S_t at discrete time step t based on which it takes an action A_t and obtains a reward R_t.

For this simple yet explanatory RL scenario, there are a few assumptions to be taken into consideration. First, there is a predefined set of possible actions A. Secondly, the time steps are discrete as in $t = 1, 2, .., n$. An action A_t is selected based on a policy π, i.e., policy π is a mapping from S_t to A_t. Each action *transitions* the environment to S_{t+1} and continues until a terminal state is reached.

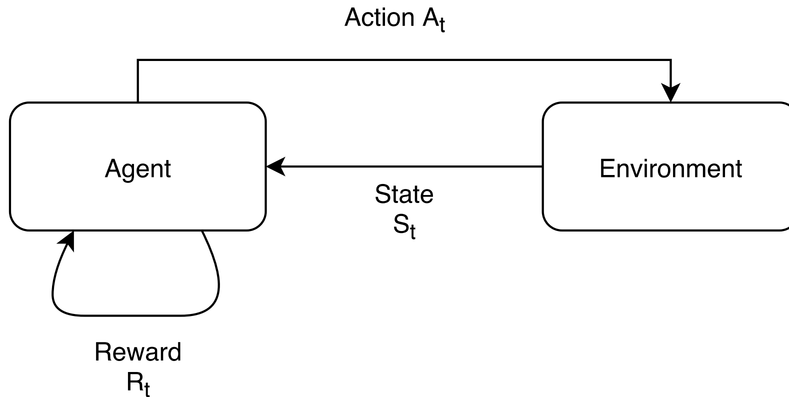


Figure 2.1: A high-level overview of a RL environment formulated as an MDP.

As stated already, the goal of the agent is to obtain the maximum cumulative reward. This can be mathematically formulated as $\sum_{t=0}^{\infty} \gamma^t r_t$. Notice the $\gamma \in (0, 1]$ term known as the discount factor. The discount factor *controls* how much the current return affects potential gains in the future, i.e., sets the preference for immediate or future rewards.

2.3.1 Model-Free Policy Gradient RL

There is an extensive family of RL algorithms and methods. In particular, we concern the model-free categorical branch as it is the common case amongst RL problems. Model-free methods fit in problems where the environment cannot be modeled explicitly, i.e., environments that do not posses a state-transition model apriori [40]. Stream scheduling Multipath-QUIC is model-free since we cannot possibly predict how an action will affect the environment. Albeit model-free approaches fit most challenges, agents in such environments are harder to train since optimization is done in a trial and error fashion.

SAILfish utilizes policy gradient methods [42] to train its policy. The goal is to learn a policy $\pi(a|s)$ where an action a_t is selected over a probability distribution based on the observed state s_t [27]. Policy gradient methods leverage function approximators to derive a policy π_θ where θ stands for the weights of the function [42]. Update of the policy parameters θ are proportional to the gradients of the obtained rewards estimated by applying the policy with respect to the parameters [27].

A descriptive taxonomy on the families and methodologies of RL approaches and methods is presented in [43].

2.3.2 Neural Networks as Function Approximators

Utilizing Neural Networks (NNs) for function approximation is a long-standing technique that has been successfully applied in many domains over the years [17, 21, 22, 27]. NNs are natural in learning meaningful representations from raw signals. Therefore, in environments with continuous variables, as in SAILfish, NNs are successful in deriving optimal approximations. Additionally, NNs do not depend on hand-engineered features [27] thus, making a strong case for RL problems.

2.3.3 The Case for Advantage Actor-Critic

There are two common methods to solve RL problems, (1) is policy optimization, and (2) value function approximation. Policy optimization regards algorithms that perform gradient ascent per step directly on an optimization objective during simulation of the environment [44]. On the other hand, value function approximation methods optimize indirectly on the objective, i.e., attempt to approximate an near-optimal policy [44]. Both methods are subjective to different flaws. In particular, policy optimization algorithms introduce variance and do not consider prior information, whilst, value function approaches, are not reliable in terms of deriving an optimal policy [44].

Combining the best of two worlds, Actor-Critic algorithms leverage value-based and policy gradient methods into a single learning scheme for optimal performance. Conceptually, the Actor learns a policy, which is inferred to select an action on the observed state. Critic on the other hand, computes the state value, i.e., an estimation of expected returns if we follow the policy moving forward. As the name suggests, Actor is the one who acts, while Critic evaluates and *proposes* changes to the Actor, based on which the policy parameters are updated.

Critic's state-value estimation is based on the expected returns [42]:

$$\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]. \tag{2.1}$$

Further, the Actor updates its network policy π_{θ_v} by computing the gradient of the policy w.r.t. parameters θ_v [45]:

$$\theta_v \leftarrow \theta_v - \alpha \sum_t \nabla_{\theta_v} log \pi_{\theta_v}(a_t|s_t) A(s_t, a_t). \tag{2.2}$$

The learning rate is represented by α, a small constant $\alpha \in (0, 1.0]$, that controls how *drastically* the policy should update per step (i.e. how big of a step). $A(s_t, a_t)$ represents

the advantage function. Conceptually, advantage serves as a baseline and helps reduce variance [45]. An estimation of advantage is provided by $r_{t+1} + \gamma V^{\pi_\theta}(S_{t+1}) - V^{\pi_\theta}(S_t)$ [27].

In a similar manner, the Critic utilizes Temporal Difference (TD) [40] to update its network parameters θ_w [27]:

$$\theta_w \leftarrow \theta_w - \alpha' \sum_t \nabla_{\theta_w} \left(r_{t+1} + \gamma V^{\pi_\theta}(S_{t+1}) - V^{\pi_\theta}(S_t)\right)^2 . \tag{2.3}$$

Finally, to mitigate the risk of converging early, [45] adds the entropy term to regularize the objective. The regularization term is estimated as $\beta \nabla_{\theta_v} H(\pi_{\theta_v}(s_t; \theta_v))$ [45] and is added to Equation 2.2 to complete the policy update. Entropy actively affects the exploration factor of the agent. Increasing constant β results in higher-exploration during training. A more detailed overview on Actor-Critic algorithms is presented in [44].

3

Related Work

Packet scheduling in multipath protocols entails more challenges than their single-path counterparts. In general, inefficient scheduling of packets might result into suboptimal performance, or even worse, might be the root cause of performance degradation. The default policy introduced in the linux kernel implementation of MPTCP [11], namely Minimum-RTT first (MinRTT), and later integrated in novel MPQUIC implementation [16], is a strong illustration of a lacking scheme that might hinder the benefits of multiple paths utilization [18, 20].

MinRTT is a simple heuristic-based scheduling mechanism that favors *fast* paths and quick dispatch of packets (i.e., round-robin packing), but its offering in simplicity, has been reported to result in head-of-line (HoL) blocking, and underutilization of paths [17, 18, 20]. This is especially true, because of the fact that MinRTT does not consider realistic network conditions (e.g. path heterogeneity, various delays in paths, etc.) but rather relies on greediness to operate.

Addressing the limitations of the default scheduler(s), researchers have been exploring alternative policies to boost performance. The majority of proposed work regards MPTCP. This is natural considering QUIC has not been widely adopted yet, and MPQUIC is fairly new and still under ongoing standardization process by the IETF [38]. In this chapter, we provide and discuss a comprehensive overview of state-of-the-art related works to SAILfish in both MPTCP and MPQUIC transport protocols.

3.1 Scheduling in MPTCP

Significant work has been conducted to address the limitations of existing scheduling mechanisms in MPTCP [17, 46, 47, 48, 49, 50, 51].

To overcome out-of-order packet blocking experienced when paths are asymmetrical, Kuhn et al. propose a scheduling algorithm that places packets for transmission based on lowest packet sequence number on the fast path, i.e., prioritizes delivery of packets with lower sequence numbers [50]. BLEST on the other hand, takes preventive measures towards HoL blocking by estimating potential packet blockage on a subflow [46].

Other recently proposed schedulers aim on different objectives. ReMP [47] proposes a scheduling scheme targeting best possible latency. To do so, Frommgen et al. leverage data redundancy to identify and utilize the fastest subflow (i.e., packet replication over all paths) [47]. MP-Dash [49] focuses on video-streaming and takes into consideration user-preference (i.e., potential path costs) and streaming deadlines to operate. [48] targets for data chunk completion time, [51] allocates packets beforehand based on path conditions, emphasizing on concurrent transmission completion (i.e., in-order arrival).

To the best of our knowledge, the work closest to ours is presented in [17]. Zhang et al. propose ReLes, a learning-based packet scheduler for MPTCP. ReLes leverages modern Deep Reinforcement Learning (DRL) algorithms and techniques [52] to learn an accurate representation of the problem space and provide optimal packet scheduling decisions. Taking into consideration the dynamicity of network environments, ReLes attempts to strike a balance between a variety of Quality of Service (QoS) objectives, and leveraging the traits of multipath protocols [17]. Evaluation of ReLes indicates significant performance benefits, especially when paths exhibit heterogeneity. In addition, ReLes demonstrates great adaptivity in response to fluctuating networking conditions based on measurements conducted in the wild. Overall, performance evaluation of ReLes fuels our motivation and provides inspiration to our work, as to the best of our knowledge, no learning-based stream scheduling mechanism for MPQUIC has been proposed yet.

3.2 The Case for MPQUIC

Recently, various efforts target the scheduling mechanism of MPQUIC to boost the performance benefits of multipath capabilities. An early work proposes a stream-aware scheduler, leveraging HTTP/2 priority-based streams aiming to enhance user experience [18]. In detail, Rabitsch et al. propose a stream-aware variant of Earliest Completion First (SA-ECF) presented initially for MPTCP [53]. SA-ECF estimates the stream completion time for a stream per path, and selects the path with the lowest RTT to transmit over. Even though SA-ECF takes into account stream priorities, the methodology of SA-ECF results into streams contesting for the fast path(s) [20].

Shi et al. propose FStream [19] focusing on loading times of high-priority streams. FStream takes into consideration path heterogeneity and stream properties (e.g., priority, stream size, etc.) to schedule streams into paths. This approach optimizes towards completion time of crucial streams but appears to be suboptimal when it comes to alleviating aggregated bandwidth of multiple paths (i.e., an important stream is scheduled into a single path) [20]. In addition, when paths are homogeneous, performance is degrading and there are no significant benefits.

In their subsequent work [20], Shi et al. utilize priority buckets to overcome prior limitations of FStream and optimize against first-rendering time of web pages. Each priority bucket holds streams with similar priorities, hence, buckets with lower priorities are scheduled for later-transmission. In other words, transmission priority is given to buckets that contain crucial-for-rendering streams. Streams can be scheduled proportionally to paths, based on stream features and in turn, paths transmit streams in a per-stream order based on descending bucket-priorities [20].

A different approach specializing in scenarios where the server knows a priori stream dependencies and determines scheduling of streams for transmission is presented in [15]. However, such a scenario is rather rare and does not conform to realistic server environments (i.e., online traffic) [20].

Despite the recent efforts, most proposed scheduling schemes for MPQUIC rely on heuristics to operate. Heuristics are often under-performing and especially in cases drawn on unrealistic scenarios and network configurations which do not conform to real world observations but rather on edge cases and sandboxed environments. Typically, schedulers based on heuristics optimize for a specific optimization objective, that does not perform well under realistic production environments [17].

It is only recently that a learning-based approach has been proposed. Wu et al. introduce a hybrid scheduling system to tackle aforementioned drawbacks of heuristic-based solutions [29]. Essentially, Wu et al. propose Peekaboo, a novel per-packet scheduling mechanism for MPQUIC. Peekaboo is a system that combines two major distinct components for achieving optimal performance: (1) a learning-based deterministic scheduling policy, and (2) a stochastic adjustment strategy [29]. To derive a deterministic policy, Peekaboo conforms to RL methods, i.e., contextual multi-armed bandit (MAB) [30], and formulates a set of actions similar to the ones introduced in BLEST [46]. In particular, the learning-based component of Peekaboo selects between a blocking action (i.e., wait for fast path to become available) and a transmit action (i.e., transmit on the slower path). Whenever both paths are available for transmission, Peekaboo relies on MinRTT to operate. Additionally, with

the purpose of tackling suboptimal scheduling decisions of the generated policy, authors of Peekaboo introduce an adjustment strategy involving stochasticity [29] as a complementary system component. Specifically, Peekaboo associates decision-making, i.e., output actions, with certain probability values. This enables on-the-fly adjustments, especially in cases where the policy does not match encountered networking conditions [29].

Performance evaluation of Peekaboo portraits significant performance gains regarding download of single-files and basic streaming scenarios in both emulated and real networking traffic. However, despite these promising results and efforts to experiment in a wide range of network configurations, Peekaboo is solely compared against single-file downloads, a rather unrealistic use-case. Particularly, Peekaboo is not measured against downloading of multiple-files, i.e., streams, and better yet, in stream prioritization schemes, e.g., actual download of a website. In addition, Peekaboo is not a pure learning-based solution but a hybrid scheme that also depends on heuristics to operate.

4

SAILfish: Scheduling Agent In Learning of Multipath-QUIC

In this chapter, we discuss the design of SAILfish, a novel stream scheduler for Multipath-QUIC. SAILfish is based on state-of-the-art RL algorithms and techniques to learn an optimal scheduling policy that aims for enhanced user QoE in web applications.

As a system, we elaborate on the overall architecture and various components. Meanwhile, focus is maintained on the novelty of our proposal, the learning-based policy. We formulate the RL problem and discuss the specifics in detail. Finally, we provide a thorough justification for the selection of the RL algorithm.

4.1 System Overview

SAILfish lays its foundations in systems design and is inherently a sophisticated networking system. The system itself aligns with the goals to alleviate performance bottlenecks of multipath protocols, and specifically that of MPQUIC. Architecturally, our system comprises of distinct components that cooperate in harmony to offer enhanced user QoE. Therefore, SAILfish is more than a derived policy. In particular, SAILfish is an online self-adjusting scheme that works against network adversities, learns from experience, and leverages advantages of multipath protocols.

In this section, we aim to shed light into SAILfish system components and provide a descriptive overview of how everything binds together. Namely, SAILfish consists of (1) MPQUIC server(s), (2) scheduling agent, and (3) Traffic Monitor. MPQUIC server is a typical HTTP server that leverages MPQUIC protocol in application layer to deliver content to clients instead of traditional TCP & TLS. In our proposed system, MPQUIC

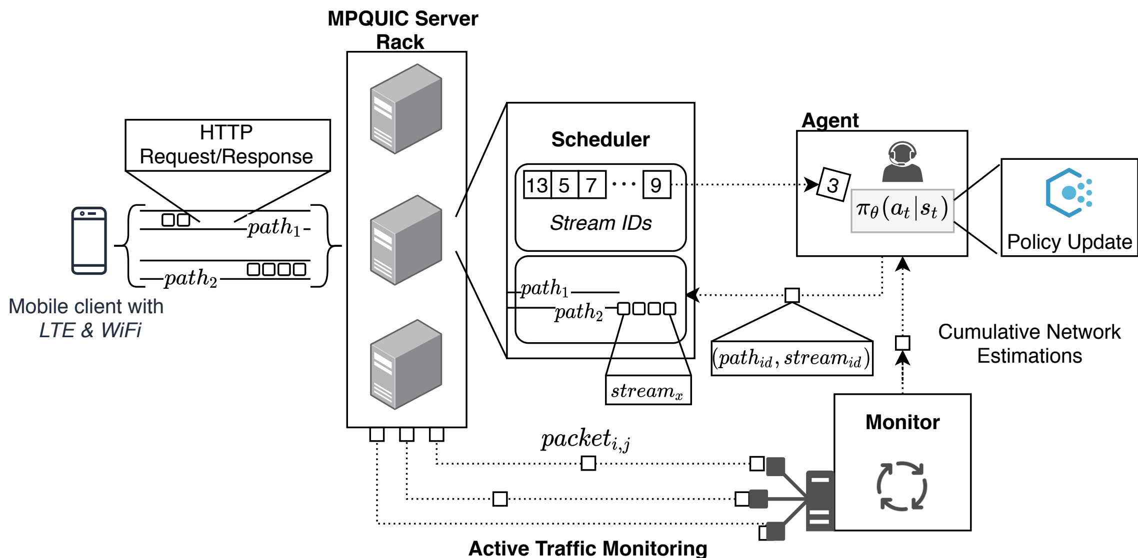


Figure 4.1: High-level overview of SAILfish.

offloads scheduler module to the agent. The agent utilizes modern RL algorithms to learn a policy that places streams to optimal paths for transmission. To do so, the Monitor component is leveraged. Traffic Monitor is responsible for providing representative network insights at given time to the agent, for the sole purpose of (1) making informed scheduling decisions, and (2) updating its policy.

Figure 4.1 depicts a deployment viewpoint of SAILfish and a realistic use-case scenario. Initially, a client requests to fetch a website from a remote server. Both the client and server support and utilize MPQUIC. The request contains prioritized Streams based on HTTP/2 weight prioritization discussed in Section 3.2. SAILfish consults the agent which in turn utilizes the observed network state from active traffic monitoring module, to provide the server with *scheduling plan* for stream-to-path transmission. The server places streams into paths as depicted by the agent policy. Finally, the server responds with HTTP message to the client transmitting the requested website.

4.1.1 Learning-based Scheduling

In Figure 4.1 the RL agent serving scheduling requests that originate from innate scheduler of MPQUIC is depicted. The requests come in a stream-based manner, i.e., each scheduling request concerns a certain stream.

The scheduling decision is based on a learnt policy. To infer the current policy, the agent dictates knowledge of specific network insights (i.e., state) at given time. In our system,

such information derives through the monitoring component which captures network activity. More specifically, there are distinct attributes that we collect and process, for instance, RTT, bandwidth estimations, and aggregated packet loss.

SAILfish scheduling policy leverages Neural Networks (NNs) as function approximators. Thus, collected network data is provided as input for model inference, which outputs the appropriate path to transmit the stream. The result of this operation is returned to MPQUIC which assigns the stream for transmission over the selected path. Eventually, the stream is packed into packets and transmitted as a response to the originator (i.e., client).

While this process is ongoing, the agent is able to adjust to network adversities and learn from new data. Essentially, our agent is capable of *self-healing* and *surviving* unforeseen circumstances that occur in dynamic networks. By aggregating collective experience from serving scheduling requests, SAILfish evolves *on the fly* and derives policies that fit more networking conditions of the environment. Regular policy updates are a counter-measure against heuristics-based solutions that cannot adapt to the dynamism of networks.

4.1.2 Tra c Monitoring

Despite the efforts to provide seamless integration, our proposed system increases module complexity and by design introduces communication overheads. Even though MPQUIC is a state-of-the-art networking protocol, it does not standardize procedures on active traffic monitoring.

To gather useful network insights, we introduce a monitoring component as displayed in Figure 4.1. Our proposed architecture illustrates a necessary monitoring module that actively collects traffic data, estimates performance metrics and provides aggregated insights to the agent as a networking state.

The state is mandated by our agent, since, it is provided as input to the policy, described in Section 4.1.1. Estimation of network metrics is constrained to the bare-minimum in order to ensure fast computation times and even faster delivery. Particularly, we are concerned in bandwidth and RTT measurements. Hence, we utilize state-of-the-art algorithms for the estimations. For bandwidth, we track delivery rate based on BBR [54], while RTT measurements rely on [55].

Yet, our design choices result in two drawbacks to consider for future work: (1) an induced performance overhead as a result of computations and especially communication between modules, and (2) tightly-coupled components, failure of monitor results in system degradation.

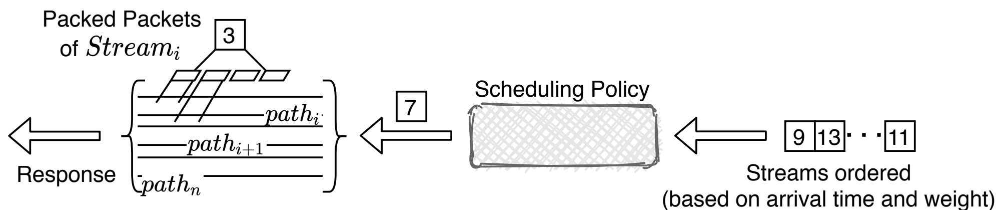


Figure 4.2: Typical heuristics stream-based scheduler in MPQUIC.

4.1.3 Limitations of the Design

SAILfish is designed to guarantee optimal scheduling policies in favor of multipath protocols and specifically targets for user experience. Therefore, by contract SAILfish architecture lacks in other quality aspects of distributed networking systems.

As a result, our proposed system is hardware constrained. This means we do not consider low-resource, low-power or limited environments. Thus, we have expanded our architecture from one monolith application-layer protocol (i.e., MPQUIC) into distinct software components. In other words, our hardware requirements have sharply increased. Each of the modules, requires state-of-the-art hardware capabilities (e.g., modern GPU for agent model inference, etc.).

4.2 Learning Scheduling Policies

To derive an optimal scheduling policy, one of the important prerequisites is the definition of an environment that meets certain criteria. We consider specific networking estimations for the agent to utilize during training. In addition, we need to formalize the action space, and motivating factor for our agent, i.e., the reward.

In this section, we formulate the RL problem in a formal manner, and specify the optimization objective. Furthermore, we present a compelling case against our selection of the RL algorithm.

4.2.1 Problem Formulation

We consider a server that utilizes MPQUIC to deliver content to clients, i.e., fetch websites. On the other end, we have a client that supports MPQUIC but packet scheduling is performed on proposed vanilla implementation [16]. Both endpoints need MPQUIC to establish and utilize a multipath QUIC connection. More details on path establishment

and secure handshake are described in [16, 38]. In the context of learning a scheduling policy and for simplicity, we focus on the server side.

The client initiates a connection and fires concurrent requests as introduced in [19]. Therefore, requests are stream-based and HTTP/2 prioritized (i.e., based on assigned priority weights). Incremental unique identifiers are assigned per stream based on arrival time, and each endpoint keeps track of its own streams.

The server orders streams on arrival and assigned priorities and marks for transmission in an iterative manner. In heuristics-based solutions [17, 19, 20], the scheduler aggregates network statistics at hand and determines a path to transmit the stream, or a portion of the stream. An example of such a scheduler, similar to state-of-the-art approaches [19] is presented in Figure 4.2.

In our approach, we utilize a similar architecture. We consider distinct scheduling epochs divided by time-steps $t \in \mathbb{N}$. Each scheduling epoch, streams available for transmission are assigned to a distinct available and failure-free $path_i$, where in our case, $i \in (0, 2]$. The target objective for our scheduling policy is to find the best possible set of tuple assignments $(path_i, stream_j)^+$, where $i, j \geq 1$.

4.2.1.1 State Space

The state space represents a snapshot of the learning environment. Snapshots capture the network statistics at time t, i.e., every scheduling epoch. Each time-step t, we opt for path assignment a set of available streams $(stream_i, stream_{i+1}, \ldots, stream_n)$, where $i = 2k + 1$ and $k \in \mathbb{N}^+$. Recall from Section 2.1.2 server-side streams are odd numbered.

Our agent observes $state_t$ in each epoch. Similarly to [17], environment $state_t$ can be presented as a set $(s_{t,1}, s_{t,2}, \ldots, s_{t,n})$, where $i \in [1, n]$ denotes the state of $path_i$. For each $path_i$, $state_{t,i}$ is a combination of the following network metrics:

- $bdw_{i,t}$ represents bandwidth estimation in epoch t
- $rtt_{i,t}$ represents smoothed RTT measurements in epoch t
- $ploss_{i,t}$ represents count of lost packets in epoch t
- $pret_{i,t}$ represents count of retransmitted packets in epoch t
- $ptotal_{i,t}$ represents count of total packets sent in epoch t

4.2.1.2 Action Space

Each $state_{t,i}$ is fed as input to the agent. As a result, agent outputs an $action_t$. Actions are immediate responses to observed environment states. This part of the process concerns advanced decision-making (recall from Section 2.3).

Actions in SAILfish represent the $path_i$ to place $stream_j$ for transmission. Since multiple streams can be allocated to different paths at the same epoch t with same $state_t$, actions can be written as a set of $action_{t,j} = (a_{t,3}, a_{t,5}, \ldots, a_{t,n})$, where $1 < j = 2k + 1 <= n$ and $k, n \in \mathbb{N}^*$, representing $stream_j$ to $path_i$ assignment.

4.2.1.3 Reward

Agents need a well-defined quantitative metric to fine-tune their policies. These metrics, otherwise known as rewards or reinforcement signals, evaluate the performance of the agent. Rewards are computed at the end of each scheduling epoch. As illustrated in Figure 2.1, every time-step t, the agent observes $state_t$ and takes $action_t$. For every action taken, $reward_t$ is calculated to assess the performance.

Therefore, reward is a sophisticated function that needs to be carefully designed as it directly impacts the optimization objective of our agent. In our case, we derive the following reward function:

$$R(s_t, a_t) = \alpha \times bdw_t^{path_i} - \beta \times stream_t^{ctime} - \gamma \times aRTT_t - \delta \times aLOSS_t \tag{4.1}$$

In detail, we formulate the agent's reward polynomial function as follows:

- $bdw_t^{path_i}$ represents the bandwidth of $path_i$ taken at $action_t$. This metric prompts the agent to prefer *faster* paths.

- $stream_t^{ctime}$ is the time difference of $stream_t$ in ms since the stream was originally requested from the client to the moment it was downloaded from the server. We penalize the agent when stream completion times increase as a quantitative measure to user experience.

- $aRTT_t$ stands for aggregated latency of both paths at time-step t and is measured in ms. The penalty on increased RTTs incentive's agent to respond to lower latency paths.

- $aLOSS_t$ stands for aggregated count of lost and retransmitted packets at time-step t. Similarly to αRTT_t we punish the agent in favor for paths that experience less packet loss.

- Finally, α, β, γ and δ are constant variables, where $c \in (0, 1.0]$. This allows for weight adjustment on the metrics, i.e., what we consider as most crucial towards our objective.

4.2.2 Learning Algorithm

4.2.2.1 A2C-based Learning

To learn an optimal scheduling policy, we propose to use an Actor-Critic methodology. The Actor-Critic variant of RL algorithms combine the best of *two worlds*. In particular, policy-based approaches favor faster convergence and perform better in stochastic environments but introduce variance [44]. On the other hand, value approximation methods are more sample efficient and stable, yet lack in reliability, i.e., guarantee for optimal results [44].

The combination of above methods introduces the powerful Actor-Critic. A high-level overview is depicted in Figure 4.3. In detail, we observe a NN with two different output layers. One is the policy, i.e., the Actor, and the other is the value approximation, i.e., the Critic. The policy represents the action to be taken at a given state, while the value approximation computes the Temporal Difference (TD) error as described in Section 2.3.3. Both Actor and Critic receive the same state as input to the NN. Further, the TD error is computed based on the reward received from the environment. Based on the TD error, both Actor and Critic update their parameters accordingly. To put it into perspective, the Critic *guides* the direction on which the Actor will update its policy. As a result, Actors have a reduced variance due to baseline reduction in gradient steps, hence, result in faster convergence [44, 45]. While Critics, are able to guarantee convergence in contrast to Critic-only methods [44].

Therefore, SAILfish leverages Advantage Actor-Critic (A2C) to learn its scheduling policy. A2C is a variant of the state-of-the-art RL algorithm Asynchronous Advantage Actor-Critic (A3C) [45].

4.2.2.2 A2C vs. A3C

The basic difference in A2C is the lack of an Asynchronous module. Essentially, A3C supports spawning multiple workers (e.g., processes, threads, etc.) each one with its own copy of the environment. Workers run training sessions in parallel and conduct asynchronous

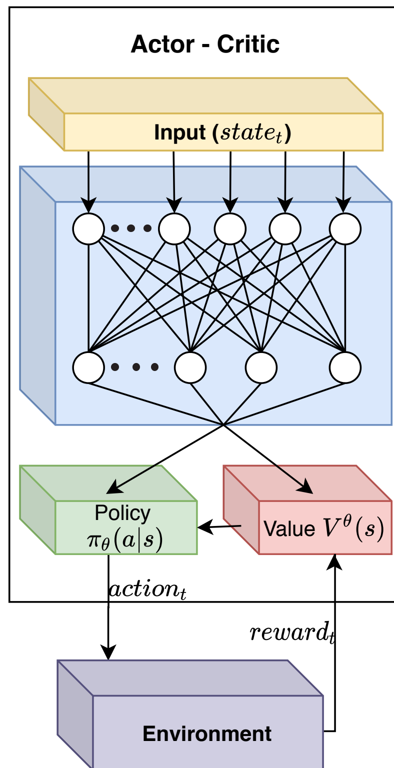


Figure 4.3: High-level overview of Actor-Critic algorithms.

parameter updates to a central network [45]. In turn, the central network maintains global parameters that sync to workers.

Meanwhile, A2C leverages one or more workers, but instead of asynchronously updating the network parameters, the *central agent* blocks until all workers communicate their progress, and conducts training on all sessions simultaneously.

Recent work indicates that the asynchronous part of A3C does not offer significant benefits [56]. On the contrary, A2C exhibits greater performance in single-GPU training environments and similar or greater performance overall [56].

To the best of our knowledge, A2C/A3C is a state-of-the-art algorithm that has been successfully applied in a variety of contexts [45, 57], including challenges that concern networking and scheduling [25, 27, 28].

5

Evaluation

SAILfish is a complex system and mandates extensive evaluation to argue its benefits. In this chapter, we elaborate on the evaluation of our proposed system. First, we proceed to discuss the implementation details of SAILfish prototype, and related challenges we encountered.

Then, we concern a formal well-defined experimentation setup to ensure correct and transparent evaluation of our system, especially in terms of performance. Further, a thorough description of findings is presented. Finally, we discuss the results and comment on our design and implementation.

5.1 Implementation

To ensure proper evaluation of SAILfish, we need to build a representative prototype. In this section, we discuss our implementation approach. Moreover, we elaborate on technical details and provide an overview of the finalised learning environment.

Based on this implementation, we train our scheduling agent to derive an optimal policy. The best performing model of our training sessions is further utilized in upcoming sections and is used as a comparison metric with other related work in the domain of MPQUIC scheduling.

5.1.1 Setting up MPQUIC

We start building our system from the most important component, that of MPQUIC. As a starting point, our implementation is based on novel MPQUIC scheduling work FStream by Shi et al. in [19]. Subsequently, Shi et al. have integrated FStream into the novel open-source implementation of MPQUIC [58] from Coninck et al. [16, 38].

In a similar manner, Coninck et al. have extended the original open-source contribution of Clemente et al. on Github[1], QUIC implementation in pure go[2]. With integration of our codebase on top of existing work, the benefits are manifold. First, our results are aligned with prior work, thus, making comparisons is easier and more malleable. Second, continuous work on fellow researchers boosts transparency and increases reproducability of results. Third, the need to reinvent the wheel is diminished, hence, productivity is enhanced and focus is shifted towards novelty.

Existing implementations of MPQUIC rely on Mininet [59, 60] to operate and especially simulate experiments. Mininet is a fully-fledged networking emulation tool. It runs on a VM on top of the host OS. Mininet creates virtual networks that are particularly useful in evaluation.

Since we base our work on the FStream implementation, a stream-based mechanism is already in place. Additionally, Shi et al. have modified MPQUIC to support prioritized streams (i.e., weighted) based on HTTP/2. Furthermore, example client and server integrations are available for testing the alterations.

Therefore, we can directly focus on the scheduling component. We have replaced the scheduler with a messaging mechanism based on the ZeroMQ protocol [61]. More specifically, instead of deriving a scheduling plan (i.e., schedule streams for transmission), the scheduler issues requests to the agent in a REQ/REP pattern, whilst encapsulating network insights (e.g., smoothed RTT, lost packets, etc.). The associated response contains the agent's plan on stream-to-path allocation, which MPQUIC follows faithfully. Meanwhile, we assume zero path failures.

5.1.2 SAILfish in Detail

SAILfish leverages NNs as function approximators for the A2C implementation. To ensure robustness, reliability and fast prototype integration, our implementation relies on the state-of-the-art open-source TensorFlow Deep Learning library [62] and TFLearn [63].

Additionally, we base our agent on the publicly available A2C implementation of Pensieve [27]. Pensieve leverages A2C to learn Adaptive-Bitrate (ABR) algorithms. To do so, Pensieve trains on network statistics and outputs ABR selection for the next video frames. The similarity of learning ABR algorithms with scheduling policies is conspicuous. By leveraging Pensieve's agent code, we save a lot of effort and moreover, aid attempts to replicate results. Furthermore, it increases the credibility of both works.

[1] www.github.com
[2] GoLang

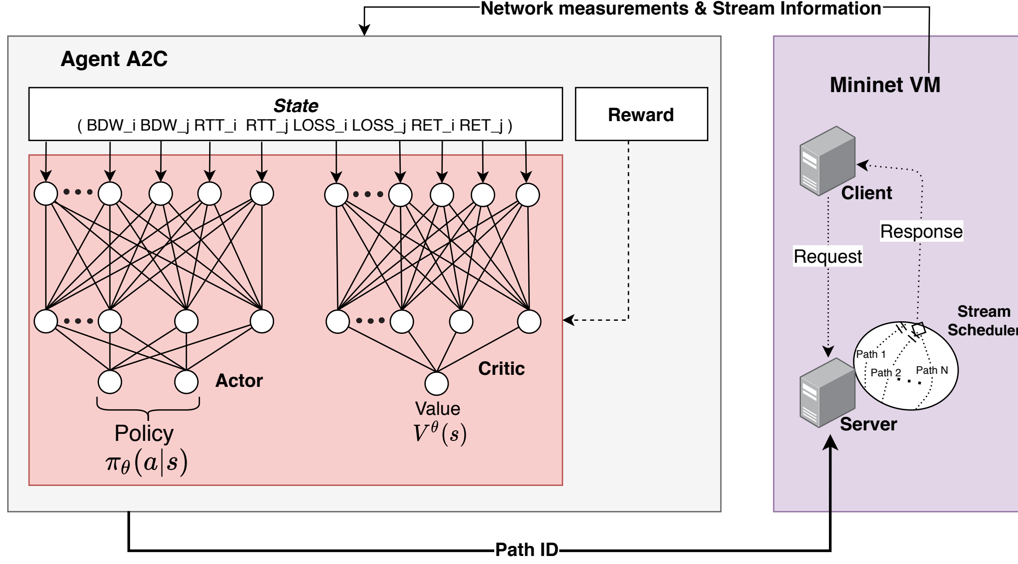


Figure 5.1: Training environment of SAILfish.

Both Actor and Critic networks have identical structure with the sole difference of the output layer. The Actor applies the softmax function for the policy output, whilst Critic directly outputs the value function, i.e., it applies no activation function. Analogous to Pensieve [27], our agent comprises of 1D-CNN input layers for past latency estimations per path (i.e., RTT). The idea to utilize CNNs for time-series stems from successful applications that argue comparable or better performance than other techniques (e.g., RNN variants, LSTM, etc.) [64, 65, 66]. The layers are merged into a 128 neurons fully-connected hidden layer [25].

We specify hyperparameters similar to the defaults of Pensieve. More specifically, we set the discount factor $\gamma \leftarrow 0.99$ and learning rates $\alpha \leftarrow 0.0001$ and $\alpha' \leftarrow 0.001$, for Actor and Critic respectively. A high discount factor indicates the agent aims for future-proof rewards instead of instant gratification. To avoid premature convergence, we set the entropy weight $\beta \leftarrow 10.0$ and slowly decrease it over time.

For the reward design described in Section 4.2.1.3, we set the constants to the reward function as follows; $\alpha \leftarrow 1.0, \beta \leftarrow 1.0, \gamma \leftarrow 0.8$, and $\delta \leftarrow 0.6$. The values of our reward design are drawn empirically. Over the course of experimenting, we witness less significance in added latency and even lesser in lost packets.

5.1.3 Module Communication

To emulate the traffic monitoring component depicted in Figure 4.1, we set up a middleware that lies in Mininet VM. The middleware serves in a twofold manner. First, middleware *subscribes* to the MPQUIC client and gathers stream insights (e.g., stream completion times, total rendering time, etc.) that forwards to the agent for evaluation. Second, it aggregates network state and forwards it to the agent to accompany the scheduling request. Then, it redirects the response back to MPQUIC server.

5.1.4 Training SAILfish

We train SAILfish with batch gradient updates. In every epoch, we follow the current policy and subsequent network updates occur in the end. As an epoch in SAILfish, distinct website delivery scenarios are utilized. In particular, we maintain a list of website dependency graphs recorded for use in SPDY evaluation [67]. Dependency graphs contain objects (i.e., streams) distinguished by priority, size, file type, etc. The graphs are associated with their actual files kept in server-side, provided in [67].

Therefore, an epoch constitutes a client-side request to fetch a website from the server. Requests are divided into distinct stream-based sub-requests with browser-based priority, e.g., Firefox [68] prioritizes on Javascript, HTML, and CSS files, and postpones everything else. When the website is downloaded, rewards are computed, and a batch update is performed. Then, we proceed to the next epoch. Hence, each epoch follows the scheduling policy of the previous one.

Approximately, SAILfish takes $7 \cdot 10^3$ epochs to converge to an optimal policy. The complete overview of the training environment is presented in Figure 5.1. In the left part, we depict the host OS and the agent component that consists of the A2C modules, a request handler to receive requests from aforementioned middleware, and a simulator to run *training scenarios*. The agent is the main coordinator concerning training.

In the right compartment, we present the Mininet VM and the client-server architecture that lies within. Notice the bi-directional communication between agent and Mininet modules. On the top part, we have aggregated networking insights and stream data, whilst, on the bottom, raw scheduling requests. Network measurements are fed to the agent, to take informed actions. Since we concern only the server for simplicity, the communication channel between agent and MPQUIC scheduler stops there. For simplicity, an explicit representation of middleware component is omitted.

	Website	No. of files
1	google	6
2	twitter	10
3	facebook	21
4	youtube	25
5	renren	35
6	dropbox	35
7	alipay	40
8	aws.amazon	47

Table 5.1: Dependency graphs used in evaluation.

5.2 Experimental Setup

In this section, we cover the design of experiments conducted to evaluate SAILfish. Particularly, we concern four aspects of control. Initially, we discuss the website dependency graphs utilized to run the experiments. We shift our attention on selected network configurations and the impact per se on performance. Then, we treat the environment upon execution of experiments. Finally, we elaborate and comment on the comparison baselines.

5.2.1 Dependency Graphs

Thoroughly discussed in Section 5.1.4, we utilize certain dependency graphs for training and evaluating SAILfish provided in [67]. Each dependency graph consists of a distinct website with its associated files that lie on the server side. Additionally, a website composes a batch sample for our system. Clients parse the file and fire prioritized requests.

To ensure integrity throughout evaluation and avoid the common problem of overfitting models, we have separated dependency graphs to training and testing, respectively. Our agent has never trained on graphs from the test set, and vice-versa. The benefits of this approach are twofold. First, we ensure correctness of results, since, our agent is unaware on specifics of the particular graph. Second, we can measure how SAILfish reacts to unforeseen scenarios similar to *in the wild* evaluations.

Moreover, Table 5.1 presents the dependency graphs used for evaluating SAILfish. In particular, and based on available graphs at hand, we select eight separate testing scenarios,

in an effort to capture all cases in testing environment and align with training sessions. Specifically, we formulate three categories based on the number of files of each graph. We consider a small number of files, Table 5.1 graphs 1-3, a medium number, Table 5.1 graphs 4-6, and finally, a large number, Table 5.1 graphs 7-8. Different files not only constitute for variance in requests (i.e., unique prioritization), but also file sizes per se, have varying impact on performance. For instance, websites that contain large files are generally slower to download and this may affect scheduling decision(s).

5.2.2 Network Configurations

Similarly to the SAILfish training environment, we utilize Mininet [60] to conduct our experiments. Mininet allows creation of virtual network interfaces, including multiple paths. For each path, we assign certain values to specific network attributes. In SAILfish, we mainly concern bandwidth, latency, and chance of packet loss.

To test across a variety of network configurations we draw uniform paired (i.e., $path_1$ and $path_2$) samples and categorize them into five buckets; (1) low bandwidth similar, (2) low bandwidth dissimilar, (3) homogeneous, (4) high bandwidth dissimilar, and (5) high bandwidth similar. Different categories allow for complete coverage in networking environments instead of assuming conditions that may or may not occur in realistic dynamic networks. This enhances validity, and boosts confidence in deployment of SAILfish in the wild.

Table 5.2 depicts the complete set of path configurations we simulate ours tests on. As noted, we select five particular cases, one per category. Every configuration is controlled, to ensure validity and realism in use-case scenarios. In detail, we constrain bandwidth values ranging between 1 and 100Mbps, latency between 0.0 and 50ms, and chances of losing a packet between 0.0 and 2.0 percent.

5.2.3 Baselines

SAILfish generates a stream-based scheduling policy, i.e., streams are assigned to distinct subflows for transmission. Therefore, a direct comparison of SAILfish with vanilla *per-packet* MinRTT scheduler in MPQUIC base implementation [16] will introduce unfairness.

To mitigate the risk, we additionally compare SAILfish against a *stream-based* adaptation of MinRTT heuristic, introduced in [19]. The *stream-based* MinRTT, leverages the same minimum latency path-selection heuristic initially presented in Chapter 2 and further dis-

	Bandwidth (Mbps)		Latency (ms)		Loss (%)	
	Path 1	Path 2	Path 1	Path 2	Path 1	Path 2
low-bdw similar	15	5	3.5	12.5	0.18	0.27
low-bdw dissimilar	38	3	17.0	9.0	1.91	0.69
homogeneous	45	45	14.0	18.5	0.33	1.75
high-bdw dissimilar	56	80	18.0	23.0	1.83	1.41
high-bdw similar	82	98	21.5	14.0	1.31	0.04

Table 5.2: Multipath configurations for network simulation.

cussed in Chapter 3. The sole difference of the two schemes is instead of assigning distinct packets to paths, the *stream-based* adaptation assigns complete streams, as in SAILfish.

This allows to draw safer conclusions on the results and leverage fairness between scheduling policies. Furthermore, our approach institutes an implicit comparison between stream and packet-based MPQUIC schedulers, setting directions for future work and exploration.

5.2.4 Evaluation Environment

Putting everything into perspective, we run experiments in a similar environment to the one used for training SAILfish. However, there are two significant differences. First, in the evaluation we only concern performance results between all three scheduling schemes presented in Section 5.2.3. Second, the vanilla per-packet scheduling MinRTT policy in the base MPQUIC implementation [16] is unaware of prioritized streams.

For the first consideration, and since we account only for performance metrics, we proceed to export an optimal model from our training sessions and introduce a new minimal component. In detail, we place a scheduling request handler inside the Mininet VM with a minimal agent representation that loads the extracted model. Instead of handling traffic requests outside of Mininet, as presented in Figure 5.1, we generate responses directly in the VM. This allows for faster experiments (i.e., less communication overhead), and seamless integration for all schemes (i.e., utilization of a single physical device). Additionally, the agent is acting only as a response handler to scheduling requests, with no further training endured.

On the other hand, we proceed to modify the vanilla MPQUIC implementation [16] to be stream-aware. More specifically, we alternate the default client component to fire requests

in HTTP/2 prioritized fashion, as in the client provided in Shi et al. implementation [19], also used for training SAILfish. In this sense, we manage to ensure fairness throughout the whole evaluation for all scheduling policies.

5.3 Performance Evaluation

After carefully designing our testing environment, we proceed to describe the actual execution of website scenarios. Specifically, we discuss in detail the procedure conducted to collect data for each scenario and per scheduling policy. Then, we present our findings and analyze performance per scheduler and in combination.

5.3.1 Collecting Scheduling Data

We automate the testing procedure with minimal Python scripts. In detail, for every scheduling policy and for every website graph presented in Table 5.1, we execute all five network configurations depicted in Table 5.2. To maintain integrity in our findings, for each network configuration we run tests in an iterative manner, specifically 10 times each.

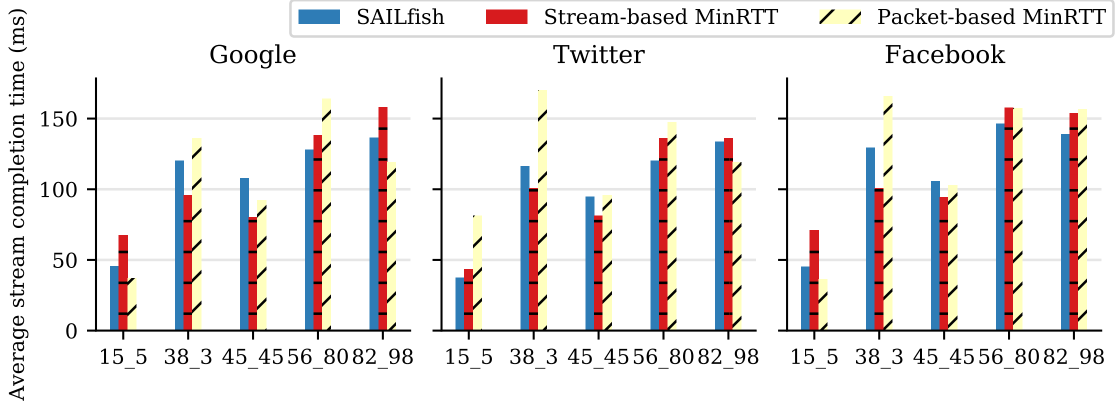
Per scenario execution, we collect a generated client log. The log contains relevant information which we utilize to analyse and discuss performance. Essentially, our logs contain detailed information on transmission per path and scheduling step (stream or packet based), completion time per stream since scheduling request fired in ms, and the total completion time to download a graph/website in seconds(s).

5.3.2 Performance Analysis

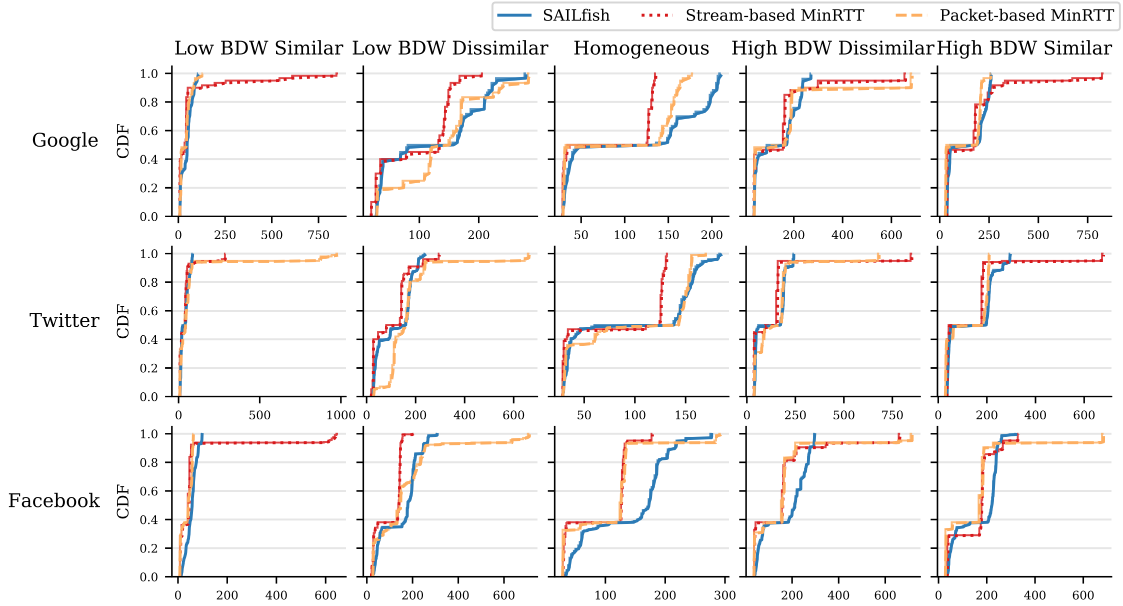
We study the performance of SAILfish by measuring aggregated stream completion times, i.e., the time it takes to download a single file from a website graph under certain networking conditions. Streams are prioritized and requested in a specific order as described in Section 5.1.4.

5.3.2.1 SAILfish vs. Stream-based MinRTT

We present the performance of SAILfish on small dependency graphs in Figure 5.2. Particularly, in Figure 5.2a we exhibit the average stream completion for each dependency graph across all scenarios and every network configuration. We observe consistently reduced stream completion times on average compared to stream-based MinRTT in environments with high bandwidth. Specifically, in high bandwidth dissimilar configuration we report



(a) Aggregated Average Stream Completion Times



(b) Cumulative Distribution Function (CDF) Plot on Stream Completion Times

Figure 5.2: Performance of SAILfish on small dependency graphs under di erent network configurations.

on average, approximately 9% reduction in completion times. Additionally, in high bandwidth similar conditions environment, a 8% decrease is reported. Yet the most significant difference in file downloading times, occurs in low bandwidth similar conditions, and is approximately 23%.

When evaluated against medium dependency graphs presented in Figure 5.3a, SAILfish performs slightly better than stream-based MinRTT in high bandwidth dissimilar configurations with an average decrease of 4%. Meanwhile, SAILfish significantly outperforms

the baseline in low bandwidth similar settings. Specifically, SAILfish manages an average 22% reduction. In addition, when paths are homogeneous a slight increase in performance is reported in two out of three cases. On the contrary, stream-based MinRTT steadily performs better than SAILfish in low bandwidth dissimilar paths setting, which increases average downloading file time by 23% in two out of three scenarios.

In large dependency graphs, SAILfish exhibits better performance than its stream-based MinRTT counterpart in most network configurations as depicted in Figure 5.4a. In detail, SAILfish displays significant performance benefits in dissimilar bandwidth conditions, with approximately 22% and 8% reduction in completion times, respectively. Moreover, our system performs slightly better in high bandwidth similar conditions. Lastly, SAILfish offers comparable performance in the remaining of scenarios.

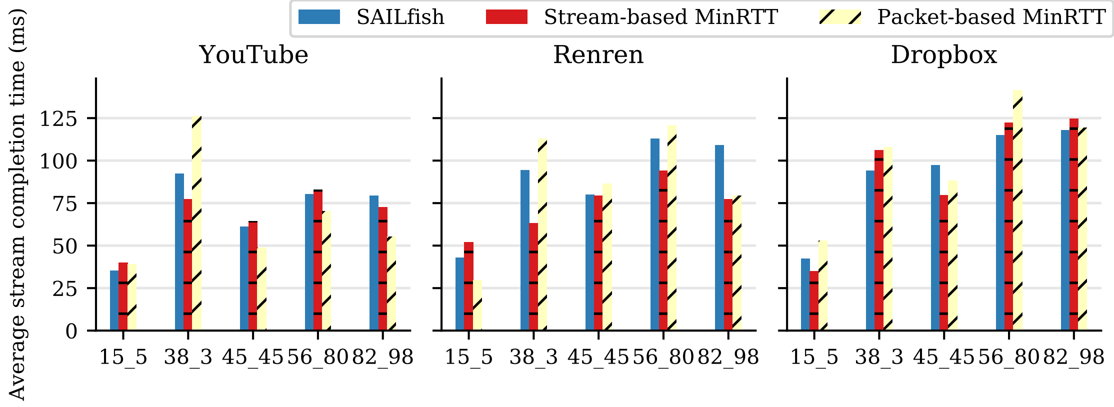
Overall, we provide tangible evidence that SAILfish outperforms or is comparable to stream-based baselines (i.e., stream-based MinRTT). In detail, we present results that validate our initial hypothesis, whether a learning stream-based scheduling policy can be derived and effectively applied to multipath protocols, and in particular, MPQUIC. Further, we exhibit that by learning a scheduling policy, we can capture the inherent variance between asymmetric paths and utilize advanced decision making to optimize scheduling decisions. Therefore, we deduct that learning-based approaches are a viable alternative to stream-based scheduling mechanisms in MPQUIC.

5.3.2.2 SAILfish vs. Per-packet MinRTT

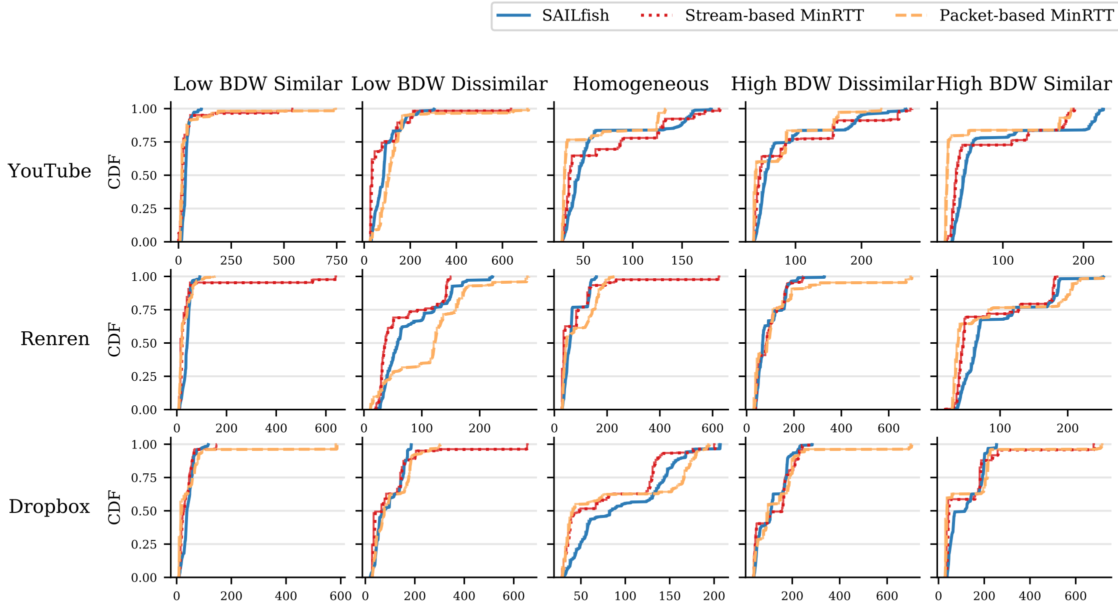
In a similar manner, we compare SAILfish to the default per-packet MinRTT scheduler implementation introduced in [11] and [16]. However, the default implementation regards per-packet scheduling decisions. In other words, a stream can be divided into multiple packets and then placed across multiple paths for transmission. Meanwhile in SAILfish, a stream and its subsequent packets are placed on a distinct path. The vital differences between these two scheduling methodologies introduce unfairness in the evaluation, while the two schemes are not comparable yet.

In particular, SAILfish exhibits better performance in low bandwidth dissimilar and high bandwidth dissimilar environments in regards to small dependency graphs in Figure 5.2a. SAILfish manages 22% and 16% decreases in average completion times, respectively. Yet, SAILfish is rarely an improvement in the homogeneous path setting.

The case is similar for medium sized dependency graphs depicted in Figure 5.3a. SAILfish constantly outperforms the per-packet scheduler in low bandwidth dissimilar conditions by an average of 18% on aggregated reduced stream completion times. On the contrary, in



(a) Aggregated Average Stream Completion Times



(b) Cumulative Distribution Function (CDF) Plot on Stream Completion Times

Figure 5.3: Performance of SAILfish on medium dependency graphs under di erent network configurations.

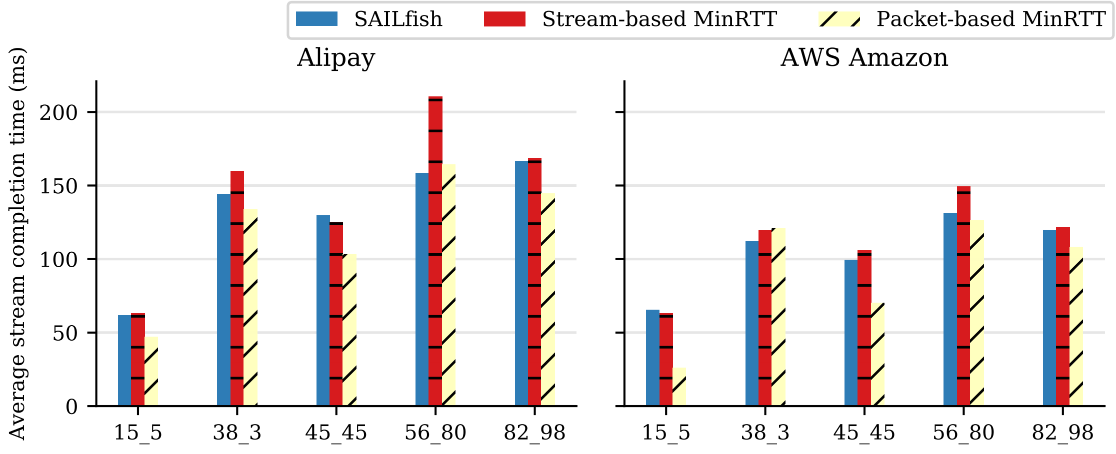
low bandwidth similar environment, SAILfish performs overall better in two out of the three cases, by 10% and 19%, respectively. The same pattern follows in high bandwidth dissimilar network state. There is a reduction of 23% and 19% accordingly. Meanwhile, SAILfish performs at best comparably in the homogeneous path, and is steadily suboptimal in high bandwidth similar conditions.

Regarding large dependency graph scenarios, we observe a notable drop in performance of SAILfish compared to per-packet MinRTT scheduler. The results are depicted in Figure

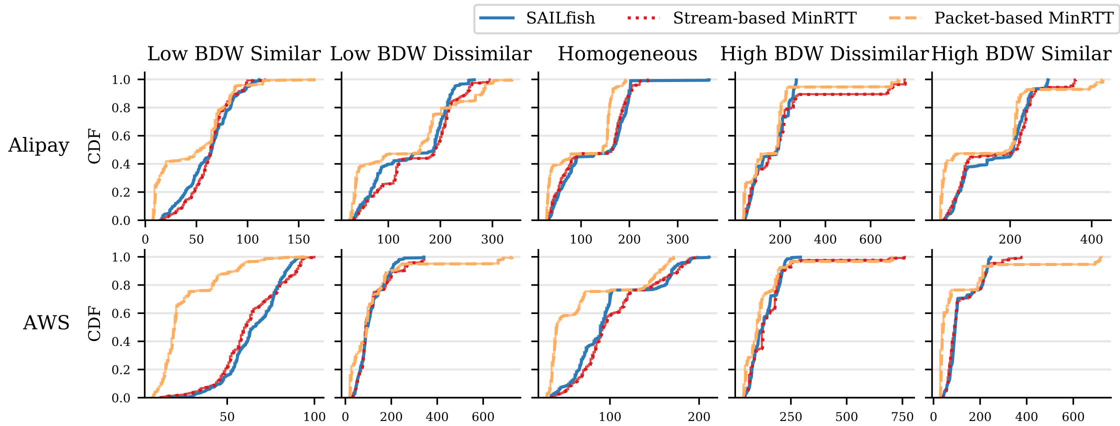
5.4a. Essentially, we record a 40% increase of averaging stream downloading times w.r.t. low bandwidth similar, homogeneous, and high bandwidth similar networking conditions. On configs with more variation (i.e., greater bandwidth, latency and loss variations), SAILfish manages comparably well to packet-based MinRTT.

Putting everything into perspective, per-packet scheduling mechanisms exhibit better performance in comparison to stream-based approaches. This is mainly due to allocation of packets between paths, in contrast with the stream-to-path mechanism SAILfish utilizes. In other words, a stream can be divided into multiple packets and then placed across multiple paths for transmission. Meanwhile in SAILfish, a stream and its subsequent packets are placed on a distinct path. Particularly, in execution scenarios discussed in Section 5.2.4, we consider scheduling of prioritized stream requests. Therefore, our environment consists of bursts of scheduling requests. In each request, the agent picks what is considered an optimal path for transmission. However, there is no consideration of consecutive stream allocation into the same path. Such an approach can result in underutilization of a less favorable path, and induce blocking if the majority of streams are assigned onto a particular subflow. On the other hand, a per-packet mechanism can avoid this type of blocking by making ad-hoc decisions whenever congestion is imminent, placing packets on the slower path to prevent performance degradation. This behavior is essentially observed in large dependency graphs, where there is a multitude of streams for transmission. In this case, SAILfish underperforms in most scenarios due to the lack of consideration of path overflow, whilst per-packet approaches assign packets onto paths more efficiently. Therefore, a comparison between a stream-based and a packet-based mechanism is not yet viable.

Additionally, in our proposed system design a learning-based per-packet policy is not currently feasible. As discussed in the implementation details we utilize our agent with minimal interference to the original MPQUIC protocol and implementation. Thus, a communication overhead between the protocol and the agent ensues. In the per-packet mechanism, such a design would be impractical. In detail, each stream is composed of a multitude of packets, while the size increases with larger streams (e.g., images). As a result, delegating all of these scheduling requests to the agent is not possible yet. In detail, and with the current environment setting such an approach would induce significant performance overheads due to communication between components, and often, the delay would result into MPQUIC shutting down the connection. On the other hand, placing the agent directly inside of MPQUIC would heavily interfere with the transport protocol, introducing significant alterations and additional performance costs. Meanwhile, such an approach



(a) Aggregated Average Stream Completion Times



(b) Cumulative Distribution Function (CDF) Plot on Stream Completion Times

Figure 5.4: Performance of SAILfish on large dependency graphs under di erent network configurations.

would constitute a centralized efficient system (i.e., multiple MPQUIC servers communicating with a single agent for scheduling decisions) incapable as each server instance would mandate its own agent to operate.

6

Discussion

In this chapter, we elaborate on the findings presented in Chapter 5 and build our case for SAILfish. Initially, a direct comparison on stream-based and per-packet MPQUIC schedulers ensues. Then, we proceed to argue the performance benefits of SAILfish in contrast to traditional heuristic-based scheduling schemes. Throughout the discussion, focus is maintained on enhanced user QoE and in particular time it takes for a client to download a website.

Further, we present and discuss certain threats to validity that affect our research. Specifically, we concern history (i.e., changes in the environment that affect the output), evaluation metrics (i.e., selection of appropriate quantitative indices), and generalization of results (i.e., generalizability in different environments). Additionally, we proceed to elaborate on how we tackle these threats to leverage integrity for our research.

6.1 Stream-based vs. Per-Packet Scheduling in MPQUIC

SAILfish offers a prioritized stream-based scheduling policy. As thoroughly described in Chapter 5, comparing a stream-based MPQUIC scheduler with a per-packet approach is not comparable yet, and introduces unfairness. However, this comparison enables us to obtain several interesting insights w.r.t. both scheduling methodologies, and gain intuition for future work.

First and foremost, we observe that SAILfish consistently performs better than baseline scheduling policy in configurations where bandwidth, latency, and packet loss chances are dissimilar, i.e., paths are *more* heterogeneous. This is visible through average reduction in stream completion times depicted in Figure 5.2a and 5.3a. On the other hand, in configurations that our environment exhibits similar traits (e.g., close bandwidth values),

MinRTT outperforms SAILfish. Especially, in case of path homogeneity, where bandwidth equals both subflows and latency is close up, SAILfish barely exhibits any progress. Taking into consideration training is conducted on diverse synthetic network traces generated on uniform distributions, and function approximators, i.e., neural networks that learn through diversity in data, the outcome seems natural. Moreover, SAILfish is not meant to perform under ideal or unrealistic network scenarios, but rather constitutes a suitable system for deployment in the wild.

Furthermore, as the size of streams in corresponding dependency graphs increases, we observe a significant drop in SAILfish's performance. This is a byproduct of per-packet scheduling approaches as presented in Section 5.3.2. As a result, stream-based schemes can be subject to suboptimal performance if allocation on multiple paths is not considered. On the other end, per-packet scheduling policies do not suffer from such limitations. Since scheduling is done at packet level, streams can be dissected into different paths making more efficient decisions, and preventing underutilization of one path. The negative impact of the stream-based scheduling approach is particularly visible in dependency graphs with lots of streams and also displayed in Figure 5.4. In this manner, we derive that stream-based approaches are not yet comparable to per-packet mechanisms, and further investigation is necessary to mitigate the risks of stream-based schemes.

6.2 SAILfish vs. Stream-based MinRTT

In contrast to per-packet schedulers, and to ensure fairness in evaluation, we compare SAILfish to MinRTT adaptation for stream scheduling mechanisms as introduced in [19]. This setting allows for equal comparison between the two schemes and enables discussions on common grounds.

To begin with, overall results indicate SAILfish is superior to stream-based MinRTT scheduler on the average completion time of streams. Essentially, SAILfish is able to outperform minimum latency approach in every high bandwidth with variant traits dependency graph. In addition, our system operates better than traditional approaches in most high bandwidth similar conditions scenarios. Both configurations, indicate clear evidence that SAILfish performs significantly better in variant conditions.

In the meantime, we observe analogous results when comparing with low bandwidth similar setting. Taking into consideration SAILfish is trained to prefer fast paths, and since one of the paths offers increased bandwidth and reduced latency, the outcome comes naturally.

Although, when conditions are heterogeneous and bandwidth is preserved at low values, SAILfish does not provide the same performance benefits across all scenarios, but only in graphs with increased number of streams. We explain this behavior as follows. The stream-based MinRTT scheduler essentially picks up on paths that offers reduced latency and places streams for transmission. However, in this particular configuration lowest latency path is also the one that offers lowest bandwidth, recall from Table 5.2, 38Mbps and 3Mbps respectively. Thus, in small dependency graphs the over utilization of one path leads to better results, as in MinRTT.

SAILfish on the other hand, continuously selects amongst both flows based on the input network state and is prone to suboptimal decisions. Yet, in larger dependency graphs we observe the opposite results. Our system is able to outperform the minimum latency path selection scheme due to the fact that placing streams on the same path eventually induces a delay in stream delivery, and does not consider multipath feature capabilities.

Another point of concern is whenever path homogeneity is encountered. As in the per-packet scheduler comparison, SAILfish is not capable of making optimal decisions in paths that feature similar traits, i.e., under equivalent conditions. We deduct that this is likely a limitation of function approximators in general, to derive and provide meaningful representations when input state is very closely related.

Putting everything together, we consider SAILfish a viable approach into tackling scheduling challenges in multiple path transport protocols. In detail, we consider the domain of DRL as a beneficial alternative to current heuristics-based schemes. An important advancement DRL offers against traditional approaches, which we leave for future work and evaluation, is the ability to adjust and improve in unforeseen network circumstances. Heuristic-based solutions offer stable performance but often result in degradation when environment assumptions do not *fit the pattern*. Another perspective we leave for future evaluation, is the design of a DRL scheduler on the premises of per-packet scheduling.

6.3 Threats to Validity

History As thoroughly described in Section 5.3, we collect our measurements by executing each scenario (i.e., dependency graph and network configuration) ten times. To avoid interference in between scenario executions, we restart the MPQUIC client and server after graph contents have been downloaded. Moreover, we re-initialize the Mininet environment with the same network configuration. In addition, whenever we proceed to execution of subsequent dependency graphs we reset the agent and the Mininet VM. By rebooting the

VM, RAM and cache memories are cleared and experiments start over on a clean slate. This ensures reliability and uniform operation of the system in measurement collections. For the SAILfish experiments, we also proceed to remove the first collection as typically the Agent request handling warm-up exhibits significant difference to subsequent measurements.

Performance Metric Reliability

We base our case on user experience and measure average completion times of each stream implying less time it takes to download a file results in overall increased user experience. Yet, this is not necessarily true in all cases. In PriorityBucket [20], Shi et al. reason the first rendering time of a web page as a vital factor for quality of experience. Other works measure stream completion times for specific streams as accurate metric [15, 18]. Thus, it is unclear whether our selection on measurement characteristics of average stream completion times, serves as a de facto root cause of elevated user experience, or other similar metrics, tend to enhance more. However, it is certain that most related works including ours focus on time measurements to argue on user QoE.

Generalization of Results

To ensure SAILfish generates a scheduling policy that generalizes well into different environments, we take two proactive measures; (1) we utilize dependency graphs not explored during training, and (2) we use diverse graphs based mainly on the number of streams. We do the same for network configurations. In this manner, we mitigate selection bias and can argue that SAILfish is able to perform well under differing conditions, i.e., being able to generalize. However, we cannot guarantee that these samples suffice when comparing to real world networking conditions and excessive number of available websites. In an ideal setting, we could assume paths are homogeneous and there are no fluctuations. Additionally, we could experiment across thousands of websites that exhibit diverse traits. Since this level of experimentation is not possible, selection bias is a threat to validity for our system.

7

Limitations

Significant efforts recently leverage learnt approaches to tackle challenges across the whole spectrum of computer systems. Despite successful applications, domains of machine and deep learning still suffer from notable limitations. In this thesis, a learning-based system is proposed focusing on scheduling policies in the domain of multipath transport protocols for the web. We contribute this chapter to discuss limitations of our approach and essentially notorious drawbacks of learnt systems.

7.1 The Why Behind The Action

In approaches based on heuristics, we can directly deduce the decision making process by taking a look on the policy or corresponding algorithm. For instance, the MinRTT scheduling policy always picks paths based on latency. Therefore, in example conditions we can derive which paths the algorithm will select, as in a deterministic environment. However, the same does not hold for complex learnt models.

Generally, ML solutions have been characterized as *black box* approaches [69, 70]. In other words, policies generated by modern DL techniques greatly lack in interpretability. This is particularly true in NNs with multiple hidden layers. Not being able to rationalize why a system responds to conditions in a certain manner, bears significant drawbacks.

First and foremost, whenever learnt approaches do not perform as intended, the process of debugging is exceptionally difficult as researchers are not aware on how agents make decisions nor what they should be looking for [69]. Another disadvantage of trained models in practice is ambiguity and raised concerns. Considering we cannot explain exactly how our system operates, we are subject to uncertainty regarding the outcome and especially in new environments. Systems that exhibit uncertainty are generally not trusted [69]. On

the other hand, utilization of simpler structures to solve complex challenges, frequently results in suboptimal performance [70].

7.2 Unexplored Territories

A significant challenge in successful applications of ML is to ensure the output model generalizes well and performs as intended to unforeseen circumstances. We train SAILfish based on simulations synthesized out of generated uniform network traces and diverse website dependency graphs. In addition, we evaluate our system on graphs that were excluded from the process of training to maintain validity.

Yet, we are still unable to precisely guarantee the performance benefits in the wild deployment of SAILfish. This is a common drawback in learnt approaches [69, 70]. No matter the diversity in training simulation environments, we cannot possibly predict every input state under real network conditions. Performance of learning-based approaches is directly affected by corresponding input. Hence, it is difficult to argue on the performance when dealing with unforeseen circumstances [69].

7.3 Computational Adversities

SAILfish constitutes for a fully-fledged networking system with various distinct components thoroughly described in Chapter 4. That being said, there is a much higher associated cost on hardware requirements than a typical heuristics-based approach. For instance, in MinRTT scheduling policy, all computation takes place in a few lines of code incorporated into MPQUIC server.

There is a whole different story for learning-based systems. As depicted in Figure 4.1, our system comprises of separate software components each responsible for different aspects of the workload. Moreover, latency matters. For example, the agent which comprises of scheduling request handling and online training, needs to reply exceptionally fast to scheduling requests. Otherwise, induced communication overhead will delay stream transmission and result into decreased user experience. It is safe to assume learnt approaches often dictate for increased computational resources.

7.4 Parameter Selection

Lastly, SAILfish, as any other ML based system is liable to hand-crafted parametric variables. Making the right combination of parameters may result into different outputs. DRL

approaches that rely on NNs to operate have a significant amount of parameters that demand fine-tuning to achieve optimal results. For instance, design of NNs and specification of the type and size of each layer might be critical to performance.

There is no de facto silver-lining and no guarantees when it comes down to specifying DRL parameters for a particular challenge. As a matter of fact, most approaches that rely on ML to operate, choose their parameters empirically. This is a strong limitation factor. There is possibly an infinite number of combinations that can be used for training a RL agent (e.g. learning rate, discount factor, etc.), but limited number of attempts.

8

Conclusion

Multipath protocols have emerged to capture network trends of the time and enhance user QoE by aggregating multiple network interfaces on modern systems. Yet recent advances introduce new challenges. In particular, multiple path protocols integrate sophisticated heuristics-based scheduling mechanisms to transmit packets w.r.t. performance benefits. However, evaluation of baselines and state-of-the-art schemes portraits the scheduling mechanism as a vital component to performance. Therefore, suboptimal scheduling policies might result in significant system degradation.

In this thesis, we propose SAILfish, a novel learning stream-based scheduling system for Multipath QUIC. SAILfish is a neural network based approach, that leverages state-of-the-art DRL algorithms to learn an efficient scheduling policy. Besides stream aware scheduling, SAILfish is a distributed networking system that is composed of certain modules, namely, (1) MPQUIC server(s), (2) Active Traffic Monitor, and (3) scheduling agent.

We contribute our system design and innate quality attributes. Moreover, we implement a prototype version of SAILfish and evaluate it against performance benefits w.r.t. per-packet and stream-based adaptation of lowest-latency baseline scheduling heuristic. Since we plan to effectively enhance user QoE, we conduct experiments on stream completion times. SAILfish performs comparably against per-packet scheduler, mainly due to significant differences between scheduling approaches, i.e., stream vs. packet-based mechanisms. On the other hand, when comparison is on the same level, SAILfish significantly outperforms stream-based heuristic baselines.

Finally, we lay the foundations for future work by discussing on the challenges SAILfish experiences and current limitations of ML systems in general. We consider an interesting approach that of a learnt packet-based scheduling system, and path capacity awareness, i.e., prevention of overloading a certain path.

References

[1] Cisco VNI. Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper, 2019. 1, 8

[2] Keith Winstein and Hari Balakrishnan. End-to-end transmission control by modeling uncertainty about the network state. In Hari Balakrishnan, Dina Katabi, Aditya Akella, and Ion Stoica, editors, *Tenth ACM Workshop on Hot Topics in Networks (HotNets-X), HOTNETS '11, Cambridge, MA, USA - November 14 - 15, 2011*, page 19. ACM, 2011. 1, 8

[3] Jim Gettys. Bu erbloat: Dark Bu ers in the Internet. *IEEE Internet Comput.*, 15(3):96, 2011. 1

[4] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM SIGCOMM 2009 Workshop on Research on Enterprise Networking, WREN 2009, Barcelona, Spain, August 21, 2009*, pages 73–82, 2009. 1

[5] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-27, IETF Secretariat, February 2020. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt. 1, 2, 7, 9

[6] Mike Belshe and Roberto Peon. SPDY Protocol. Internet-Draft draft-mbelshe-httpbis-spdy-00, IETF Secretariat, February 2012. http://www.ietf.org/internet-drafts/draft-mbelshe-httpbis-spdy-00.txt. 1, 10

[7] Preethi Natarajan, Paul Amer, Jonathan Leighton, and Fred Baker. Using SCTP as a Transport Layer Protocol for HTTP. Internet-Draft draft-

natarajan-http-over-sctp-00, IETF Secretariat, November 2008. http://www.ietf.org/internet-drafts/draft-natarajan-http-over-sctp-00.txt. 1

[8] Ryan Hamilton Alyssa Wilk and Ian Swett. A QUIC update on Google's experimental transport, 2015. 1, 7, 8, 10

[9] Lucas Pardue, 2020. [link]. 1

[10] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, RFC Editor, March 2020. 1, 10

[11] Christoph Paasch, Sebastien Barre, et al. Multipath TCP implementation in the Linux kernel. Available from http://www.multipath-tcp.org, 2017. 1, 2, 11, 17, 38

[12] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 181–194, New York, NY, USA, 2014. Association for Computing Machinery. 1, 10

[13] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, August 2011. 1, 10

[14] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, CellNet '12, page 31–36, New York, NY, USA, 2012. Association for Computing Machinery. 1, 10

[15] Jing Wang, Yunfeng Gao, and Chenren Xu. A Multipath QUIC Scheduler for Mobile HTTP/2. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking, APNet 2019, Beijing, China, August 17-18, 2019*, pages 43–49. ACM, 2019. 1, 2, 19, 46

[16] Quentin De Coninck and Olivier Bonaventure. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 160–166. ACM, 2017. 1, 2, 9, 10, 11, 12, 17, 24, 25, 29, 34, 35, 38

[17] Han Zhang, Wenzhong Li, Shaohua Gao, Xiaoliang Wang, and Baoliu Ye. ReLeS: A Neural Adaptive Multipath Scheduler based on Deep Reinforcement Learning. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1648–1656. IEEE, 2019. 2, 3, 14, 17, 18, 19, 25

[18] Alexander Rabitsch, Per Hurtig, and Anna Brunström. A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths: Paper # XXX, XXX pages. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ@CoNEXT 2018, Heraklion, Greece, December 4, 2018*, pages 29–35. ACM, 2018. 2, 17, 18, 46

[19] Xiang Shi, Lin Wang, Fa Zhang, and Zhiyong Liu. FStream: Flexible Stream Scheduling and Prioritizing in Multipath-QUIC. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*, pages 921–924. IEEE, 2019. 2, 19, 25, 29, 34, 36, 44

[20] Xiang Shi, Fa Zhang, and Zhiyong Liu. PriorityBucket: A Multipath-QUIC Scheduler on Accelerating First Rendering Time in Page Loading. In *e-Energy '20: The Eleventh ACM International Conference on Future Energy Systems, Virtual Event, Australia, June 22-26, 2020*, pages 572–577. ACM, 2020. 2, 17, 18, 19, 25, 46

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012. 2, 14

[22] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. 2, 14

[23] Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012. 2

[24] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 256–269, 2019. 2

[25] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource Management with Deep Reinforcement Learning. In Bryan Ford, Alex C. Snoeren, and Ellen W. Zegura, editors, *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*, pages 50–56. ACM, 2016. 2, 28, 31

[26] Nathan Jay, Noga H. Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 3050–3059, 2019. 2

[27] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 197–210. ACM, 2017. 2, 13, 14, 15, 28, 30, 31

[28] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural Adaptive Content-aware Internet Video Delivery. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 645–661. USENIX Association, 2018. 2, 28

[29] H. Wu, Ö. Alay, A. Brunstrom, S. Ferlin, and G. Caso. Peekaboo: Learning-based Multipath Scheduling for Dynamic Heterogeneous Environments. *IEEE Journal on Selected Areas in Communications*, pages 1–1, 2020. 3, 19, 20

[30] Tyler Lu, Dávid Pál, and Martin Pal. Contextual Multi-Armed Bandits. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, 9 of *JMLR Proceedings*, pages 485–492. JMLR.org, 2010. 3, 19

[31] Jim Roskind. QUIC: Quick UDP Internet Connections, 2013. 7, 8, 9

[32] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Oper. Syst. Rev.*, 42(5):64–74, 2008. 8, 11

[33] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In Jon Crowcroft, editor, *Proceedings of the ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, pages 24–35. ACM, 1994. 8

[34] Liansheng Tan, Cao Yuan, and Moshe Zukerman. FAST TCP: fairness and queuing issues. *IEEE Commun. Lett.*, 9(8):762–764, 2005. 8

[35] Somak R Das. *Evaluation of QUIC on web page performance*. PhD thesis, Massachusetts Institute of Technology, 2014. 10

[36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery. 10

[37] P. Biswal and O. Gnawali. Does QUIC Make the Web Faster? In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2016. 10

[38] Quentin De Coninck and Olivier Bonaventure. Multipath Extensions for QUIC (MP-QUIC). Internet-Draft draft-deconinck-quic-multipath-04, Internet Engineering Task Force, March 2020. Work in Progress. 10, 17, 25, 29

[39] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. In Chadi Barakat, Renata Teixeira, K. K. Ramakrishnan, and Patrick Thiran, editors, *Conference on emerging Networking Experiments and Technologies, CoNEXT '12, Nice, France - December 10 - 13, 2012*, pages 1–12. ACM, 2012. 11

[40] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 12, 13, 15

[41] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., USA, 1st edition, 1994. 12

[42] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1057–1063. The MIT Press, 1999. 13, 14

[43] Atrisha Sarkar. A Brandom-ian view of Reinforcement Learning towards strong-AI. *CoRR*, abs/1803.02912, 2018. 13

[44] Vijay R. Konda and John N. Tsitsiklis. Actor-Critic Algorithms. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 1008–1014. The MIT Press, 1999. 14, 15, 27

[45] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings*

of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, 48 of *JMLR Workshop and Conference Proceedings*, pages 1928–1937. JMLR.org, 2016. 14, 15, 27, 28

[46] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*, pages 431–439. IEEE Computer Society, 2016. 17, 18, 19

[47] Alexander Frömmgen, Tobias Erbshauser, Alejandro P. Buchmann, Torsten Zimmermann, and Klaus Wehrle. ReMP TCP: Low latency multipath TCP. In *2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, May 22-27, 2016*, pages 1–7. IEEE, 2016. 17, 18

[48] Yihua Ethan Guo, Ashkan Nikravesh, Zhuoqing Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating Multipath Transport Through Balanced Subflow Completion. In Kobus van der Merwe, Ben Greenstein, and Kannan Srinivasan, editors, *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking, MobiCom 2017, Snowbird, UT, USA, October 16 - 20, 2017*, pages 141–153. ACM, 2017. 17, 18

[49] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In Athina Markopoulou, Michalis Faloutsos, Vyas Sekar, and Dejan Kostic, editors, *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies, CoNEXT 2016, Irvine, California, USA, December 12-15, 2016*, pages 129–143. ACM, 2016. 17, 18

[50] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. DAPS: Intelligent delay-aware packet scheduling for multipath transport. In *IEEE International Conference on Communications, ICC 2014, Sydney, Australia, June 10-14, 2014*, pages 1222–1227. IEEE, 2014. 17, 18

[51] Hang Shi, Yong Cui, Xin Wang, Yuming Hu, Minglong Dai, Fanzhao Wang, and Kai Zheng. STMS: Improving MPTCP Throughput Under

Heterogeneous Networks. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 719–730. USENIX Association, 2018. 17, 18

[52] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 48 of *JMLR Workshop and Conference Proceedings*, pages 2829–2838. JMLR.org, 2016. 18

[53] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 147–159. ACM, 2017. 18

[54] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):20–53, 2016. 23

[55] Van Jacobson. Congestion avoidance and control. In Vinton G. Cerf, editor, *SIGCOMM '88, Proceedings of the ACM Symposium on Communications Architectures and Protocols, Stanford, CA, USA, August 16-18, 1988*, pages 314–329. ACM, 1988. 23

[56] Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. OpenAI Baselines: ACKTR & A2C, 2017. 28

[57] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. FeUdal Networks for Hierarchical Reinforcement Learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 70 of *Proceedings of Machine Learning Research*, pages 3540–3549. PMLR, 2017. 28

[58] Quentin De Coninck. Multipath-QUIC, 2017. 29

[59] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In Geoffrey G. Xie, Robert Beverly, Robert Tappan Morris, and Bruce Davie, editors, *Proceedings of the 9th ACM Workshop on Hot Topics in Networks. HotNets 2010, Monterey, CA, USA - October 20 - 21, 2010*, page 19. ACM, 2010. 30

[60] Bob Lantz et al. Mininet: Rapid Prototyping for Software Defined Networks, 2020. 30, 34

[61] Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013. 30

[62] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org. 30

[63] Yuan Tang. TF.Learn: TensorFlow's High-level Module for Distributed Machine Learning. *CoRR*, abs/1612.04251, 2016. 30

[64] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *CoRR*, abs/1803.01271, 2018. 31

[65] A. Tsantekidis, N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj, and A. Iosifidis. Forecasting Stock Prices from the Limit Order Book Using Convolutional Neural Networks. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, 01, pages 7–12, 2017. 31

[66] Anastasia Borovykh, Sander Bohte, and Cornelis W. Oosterlee. Conditional Time Series Forecasting with Convolutional Neural Networks, 2017. 31

[67] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is SPDY? In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2014. 32, 33

[68] Mozilla. Firefox Browser, 2020. 32

[69] Ying Zheng, Ziyu Liu, Xinyu You, Yuedong Xu, and Junchen Jiang. Demystifying Deep Learning in Networking. In Mosharaf Chowdhury and Kun Tan, editors, *Proceedings of the 2nd Asia-Pacific Workshop on Networking, APNet 2018, Beijing, China, August 02-03, 2018*, pages 1–7. ACM, 2018. 47, 48

[70] Arnaud Dethise, Marco Canini, and Srikanth Kandula. Cracking Open the Black Box: What Observations Can Tell Us About Reinforcement Learning Agents. In *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, pages 29–36. ACM, 2019. 47, 48