Enhancing QUIC: Quality-of-Service LLM Serving and MASQUE Proxy Scheduling

Rithvik Chuppala

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-64
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-64.html

May 8, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Enhancing QUIC: Quality-of-Service LLM Serving and MASQUE Proxy Scheduling

Rithvik Chuppala

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sylvia Ratnasamy
Research Advisor

May 3, 2024

(Date)

* * * * * * *

Professor Scott Shenker
Second Reader

4/30/2024

(Date)

Enhancing QUIC: Quality-of-Service LLM Serving and MASQUE Proxy Scheduling

by

Rithvik Chuppala

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sylvia Ratnasamy, Chair
Professor Scott Shenker

Spring 2024

Enhancing QUIC: Quality-of-Service LLM Serving and MASQUE Proxy Scheduling

Copyright 2024
by
Rithvik Chuppala

Abstract

Enhancing QUIC: Quality-of-Service LLM Serving and MASQUE Proxy Scheduling

by

Rithvik Chuppala

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

This thesis explores the QUIC network protocol, a transport-layer protocol positioned as TCP's successor in modern network architectures. Focusing on QUIC's key design aspect of stream scheduling, this research investigates two contemporary networking applications: Large Language Model Serving and Network Proxying. The first chapter presents various stream scheduling algorithms tailored to the unique demands of LLM serving, providing novel approaches to optimize data transmission and model service in resource-constrained environments. In the second chapter, this thesis demonstrates the role of stream scheduling in the context of MASQUE proxies, exploring ways to improve the performance and scalability of QUIC-based tunneling protocols. As new applications demand custom-tailored network functionalities, stream scheduling emerges as a fundamental primitive for delivering application-specific optimizations, blurring the lines between the end-host and the network infrastructure. At its core, QUIC itself departs from traditional conventions, relying on the plain datagram abstraction and assuming the responsibilities of reliable delivery, security, and application-level semantics, integrating Layers 4 through 7 in the OSI model. This paradigm shift emphasizes the importance of co-designing protocols and algorithms for application semantics. This work aims to enhance the efficacy of stream scheduling in QUIC, addressing the evolving demands of modern networking applications.

To my mother Vineetha and father Ravi

Contents

Contents ii

List of Figures iii

List of Tables iv

1 Quality-of-Service Aware LLM Serving 1

2 Scheduling in MASQUE Proxies 15
- 2.1 Introduction . . . 16
- 2.2 Background . . . 18
- 2.3 Implementation . . . 23
- 2.4 Evaluation . . . 24
- 2.5 Future Work . . . 33
- 2.6 Conclusion . . . 34

Bibliography 35

List of Figures

2.1 Apple's Dual-Hop Relay Architecture 17
2.2 Example HTTP/3 CONNECT-UDP Request 19
2.3 Tunnel Packet Encapsulation and Decapsulation 19
2.4 Tunnel Connection Architecture 20
2.5 Tunnel Connection Architecture for Single Client, Single Proxy, and Multiple Targets 21
2.6 Completion Rate of Tunneled Streams using RR vs FCFS Scheduling (8s deadline) 25
2.7 RTT Latencies of Tunneled Streams using RR vs FCFS Scheduling 26
2.8 Overal Throughput of 100 Tunneled Streams using RR vs FCFS Scheduling . . 27
2.9 RTT Latencies of Nested Streams using ABS, RR, and FCFS Scheduling 29
2.10 RTT Latencies of PIFO ABS vs. Base ABS Scheduling 31
2.11 Stream Management Overhead: PIFO ABS vs. Base ABS 32

List of Tables

Acknowledgments

I would like to express sincere gratitude to my advisor Professor Sylvia Ratnasamy for providing me the opportunity to pursue research as a Masters student and her unwavering patience, guidance, and support throughout this journey. I would also like to acknowledge Professor Scott Shenker for taking the time to be the second reader for this thesis. I would like to express special gratitude to my collaborator Siddharth Jha, without whom the first Chapter of this thesis would not have been possible. I would also like to express my thanks to the following faculty and students who have guided this work and enriched my research experience at Berkeley: John Kubiatowicz, Christopher Fletcher, and Silvery Fu. Special thanks to my parents, grandparents, aunt, friends, and good friend for providing me day-to-day support and motivation for the past five years and throughout my academic journey. Finally, I would like to thank God.

Chapter 1

Quality-of-Service Aware LLM Serving

In this first chapter, we introduce the QUIC network protocol and the notion of application-level stream scheduling in QUIC. We design, implement, and integrate stream scheduling semantics in a reference QUIC implementation and benchmark our schedulers. Asserting that the future of Large Language Models - still a nascent space - will necessitate catering to various classes of requests, we apply our scheduling semantics to deliver Quality-of-Service for serving LLMs. To this end, we introduce and evaluate novel scheduling algorithms at the network-level in conjunction with a best-effort serving system that employs deep Q-learning to adjust model quality based on task distribution and system load. Our end-to-end model-serving system effectively caters to QoS differentiation between requests and outperforms current model-serving standards in per-token latency and response quality metrics.

Quality-of-Service Aware LLM Serving

Abstract

Many applications must provide low-latency large language model (LLM) service to users or risk unacceptable user experience. However, over-provisioning resources to serve models is often prohibitively expensive. In this work, we identify various classes of LLM requests, each with different quality-of-service requirements. To best meet the requirements of all traffic classes, we design and implement scheduling algorithms for the QUIC network protocol. Moreover, we present a best-effort serving system that employs deep Q-learning to adjust service quality based on task distribution and system load. We train the Q-learning agent to optimize latency and model serving quality of higher priority classes while achieving fairness and best-effort response quality for lower priority requests. Our network schedulers show better request latency and completion rate performance compared to the standard QUIC protocol, as well as TLS/TCP. Overall, our end-to-end model-serving system effectively caters to QoS differentiation between requests and outperforms current model-serving standards in per-token latency and response quality metrics.

1. Introduction

Applications in the last decade have evolved from using machine learning in background functions such as data analytics and monitoring to being at the forefront of user experience. Over the past couple of years, many applications have adopted large language models (LLMs) to provide users with both custom and interactive experiences. The need for latency guarantees is critical for such applications as services cannot simply hang and become unavailable to users. LLM systems are faced with the challenge of serving a wide range of user demands, such as varying context lengths and request arrival rates, while simultaneously meeting application QoS requirements. This necessitates an architectural tradeoff as the simple solution of over-provisioning resources to serve dynamic application needs is prohibitively expensive for small businesses and independent developers.

The status quo for applications is to serve queries using a model of fixed quality. For example, within the OPT model family (Zhang et al., 2022), there are models of various sizes, with larger models providing higher-quality responses. An application may choose OPT-6.7B to serve all queries, with this model replicated and/or partitioned across a set of GPUs. We explore a new paradigm of best-effort serving in which models of different quality and latency exist simultaneously in the serving system with a router that sends client requests to each model. For example, suppose an application would like to support OPT-6.7B quality when possible. However, during periods of higher-than-expected request rates, the system will choose to serve at OPT-1.3B's quality in order to maintain responsiveness. During even more contested periods, the system may serve queries at OPT-125M's quality in order to stay available. Serving these smaller models in conjunction with the large model is possible since smaller models consume less device memory than the large model, in the same way speculative inference (Leviathan et al., 2023) has become an option for low-latency model serving.

In this best-effort setting, the serving system must serve at the highest possible quality while maintaining availability. Simply serving at the smallest model's quality all the time will be undesirable for a user, even though availability would be high. In this work, we show that routing queries to models is dependent on the set of tasks, the distribution of those tasks, and the load on the system. In order to learn a router that efficiently routes client requests to LLMs while meeting latency guarantees, we utilize deep reinforcement learning (RL) techniques with minimal hyper-parameter tuning.

State-of-the-art model serving systems utilize REST API/gRPC endpoints over base HTTP/2 (TCP/TLS) network protocols (Agarwal et al., 2023) (vLLM Team, 2023) (RayTeam, 2023). However, model-serving architectures are agnostic to application-specific network patterns and do not cater to QoS differentiation between requests. While most active research explores major bottlenecks in the speed of model inference, we find that effective network scheduling not only allows us to cater to different traffic classes but also meet per-token latency deadlines during periods of high request load and network resource contention. For example, one area of ongoing exploration is the use of LLMs in synthesizing large text corpora, such as collections of legal documents (Bornstein & Radovanovic, 2023). Common uses of the widely popular LLM platform, ChatGPT, involve providing the LLM with contextual information fol-

lowed by a short question prompt (Raf, 2023). The former potentially requires higher model qualities to accurately synthesize domain-specific technical language, whereas the latter needs low latency of response to provide a fluid and interactive user experience.

To meet request SLOs, we leverage one of the key design components of the QUIC network protocol - multiplexed connection streams. This allows us to send different classes of data in separate streams, reducing multi-connection overheads. To efficiently manage these streams and meet the variety of application requirements detailed above, we propose the implementation of a scheduling abstraction in open-source QUIC. By considering LLM traffic at both the network level and model-serving level, we are able to schedule and serve requests at low latencies even in the presence of wide fluctuations in client behavior.

In summary, we make the following contributions in this work:

1. We implement QUIC stream schedulers for client requests and responses to meet the QoS requirements of various traffic classes
2. We train and employ an RL router agent to send client queries to appropriate models in order to maximize response quality while meeting user-defined latency guarantees
3. We design and implement an end-to-end model-serving framework utilizing both network stream scheduling as well as dynamic model selection
4. We evaluate our system, analyzing LLM application requirements, and running request loads to benchmark against existing model-serving techniques.

2. Background and Related Work

2.1. Large Language Models

LLMs have emerged as a powerful service for modern applications. There is a wide spectrum of LLMs which forms a trade-off of quality and latency. Larger models with more parameters can serve client requests at a higher quality but incur higher latency. There is also a wide spectrum of tasks that can be served using LLMs. Such tasks include summarization (Hermann et al., 2015; Narayan et al., 2018), translation (Cettolo et al., 2017), question answering (Rajpurkar et al., 2016), etc. Certain tasks are easier than other tasks in that the quality loss of serving that task with a smaller model as opposed to a larger model is small. As a result of this observation, speculative decoding (Leviathan et al., 2023) has emerged as a popular solution that uses a smaller draft model to speculatively generate tokens that later are verified in parallel by a large model. However, serving speculative inference in real deployments with continuous batching techniques (Yu et al., 2022) is still an open problem and common speculative inference implementations still use a batch size of one (Leviathan et al., 2023) (Kim et al., 2023).

2.2. LLM Serving

Prior work on LLM serving (Li et al., 2023; Zhang et al., 2023; Gujarati et al., 2020) assumes that client requests are bound to a specific model. Our best-effort approach relaxes this, allowing for increased scalability. Big Little Decoder is a lossy speculative inference technique (Kim et al., 2023) that allows clients to adjust quality and latency by changing hyper-parameters. However, this requires a hyper-parameter search for every task, does not adjust under load, and struggles from the practical issues of speculative decoding as mentioned prior. Autoscalers such as Ray (Moritz et al., 2018) dynamically increase GPU instances under load. However, acquiring on-demand GPU instances is expensive and not instantaneous.

2.3. Deep Reinforcement Learning

Deep RL is a promising technique for learning to control systems and has been successfully applied in a variety of areas such as continuous controls (Brockman et al., 2016) and games (Mnih et al., 2013) . There are three core components in any RL problem: states, actions, and rewards. Given the state, the RL policy chooses an action, which gives it a reward for that action and transitions the environment to the next state. The goal of RL algorithms is to maximize the total rewards seen by the policy as it takes actions and transitions to different states. Deep Q-learning methods learn a Q-function, represented as a neural network, that map state-action pairs to the expected return of taking the action in the state and then following the policy. After fitting the Q-function of the optimal policy, the Q-function may be used to select actions with the highest expected reward. Popular algorithms in this area include DQN (Mnih et al., 2013), Double Q-learning (Van Hasselt et al., 2016), and PER (Schaul et al., 2015).

2.4. QUIC

The QUIC network protocol (standardized in IETF RFC 9000, 9001, 9002) is a transport-level protocol built on UDP and offers endpoint-to-endpoint uni- or bi-directional connections, reliability semantics, congestion control mechanisms, encryption/security via TLS, low-latency connection establishment, and notably, stream multiplexing (Iyengar & Thomson, 2021) (Thomson & Turner, 2021) (Iyengar & Swett, 2021). Since its release by Google in 2013, QUIC has gained wide adoption across web browsers and today accounts for all of Google's frontend server traffic as well

as 10% of global internet traffic (Langley et al., 2017). The new HTTP/3 standard integrates QUIC into the HTTP stack, substituting TLS/TCP in HTTP/2 (Bishop, 2022). Real-time applications currently leverage UDP to avoid delays from retransmissions for applications such as video calls and DNS. QUIC, which builds on UDP, also offers the base datagram network abstraction for such applications (Pauly et al., 2022). QUIC has found its primary niche in the HTTP/3 stack for web applications as a replacement for TCP/TLS in HTTP/2 (Bishop, 2022). However, its properties make it a compelling network protocol in several other use cases.

2.5. QUIC Streams

One of the key design components of the QUIC protocol is the use of time-multiplexed streams in a single point-to-point connection. While HTTP/2 (built over TLS/TCP) attempts intra-connection multiplexing, it suffers from head-of-line blocking, where all streams are blocked if just a single stream experiences packet loss (Langley et al., 2017).

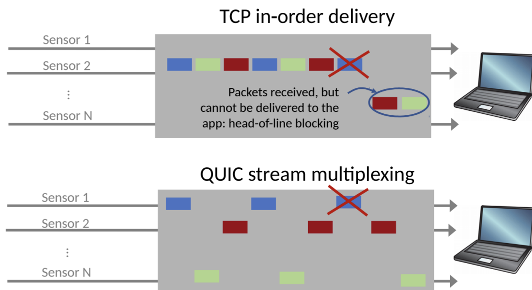

Figure 1. Stream multiplexing in TCP (which suffers from head-of-line blocking) vs QUIC. Credits to (Chiariotti et al., 2021).

QUIC resolves this through flow multiplexing and retransmission semantics at a stream-level granularity. In TCP, retransmission and flow control semantics are maintained at the connection level, whereas in QUIC, these are provided at the stream level. As such, packets in a stream do not block or wait on packets of another stream in the context of loss recovery.

Priority for stream multiplexing was first introduced to HTTP/2 in RFC 7540 (Belshe et al., 2015). However, it was subsequently removed in HTTP/3 (Bishop, 2022) and the revised HTTP/2 (Thomson & Benfield, 2022) due to high complexity and low feature utilization. As an artifact of the old HTTP/2 priority scheme, the QUIC RFC recommends that implementations offer ways to indicate the relative priority of streams (Iyengar & Thomson, 2021); however, a wide number of open-source implementations do not provide this feature.

2.6. Stream Scheduling

Despite a stream multiplexed design, the QUIC RFC does not define or suggest stream scheduling behaviors. Consequently, the vast majority of open-source QUIC implementations use either a default First-Come-First-Serve (FCFS) or Round Robin (RR) scheduler (Kutter & Jaeger, 2022). These implementations do not provide rich scheduling abstractions beyond the default. Literature exploring stream scheduling in QUIC is also quite limited. One work proposed an abstraction to allow applications to effectively map information flows to QUIC streams, focusing on the ability to define correlated data flows (Chiariotti et al., 2021). Another work implements the MPEG-DASH protocol over QUIC, involving scheduling only within the context of the DASH protocol (Cui et al., 2022). Fernandez et al. explore modifications of the QUIC stream scheduler to provide QoS and latency guarantees to UAV video and control flows (Fernández et al., 2023). They focus on priority scheduling, which some open-source implementations already provide upon RFC recommendation. To the best of our knowledge, there are no existing implementations of non-priority-based scheduling algorithms, nor intricate QoS/multi-level scheduling paradigms in open-source QUIC.

3. Design and Architecture

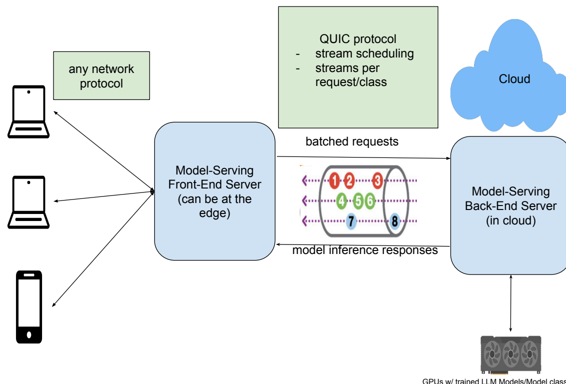

Figure 2. Architecture Diagram of the system. Front-end servers receive client requests, initiate QUIC streams with the model-serving back-end, and schedule request-response flows. The back-end model serving environment is detailed in Figure 3.

Interactive applications should aim to query as large of an LLM as possible while still meeting an acceptable deadline requirement for their user. However, just using one model type (e.g. OPT-175B) will lead to unacceptably high latencies and impossible-to-meet deadlines during periods of high request load to the inference system. In order to cope with the increasing demands, LLM serving systems need a

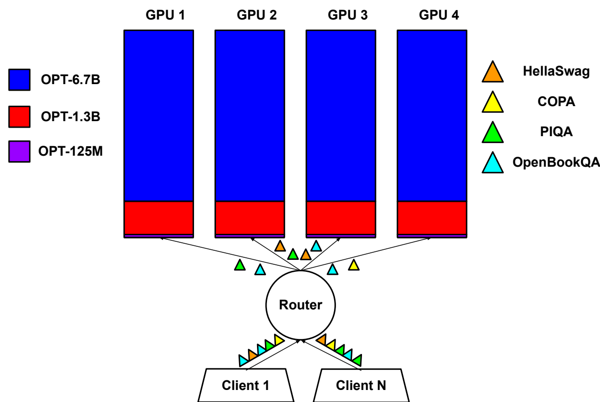

Figure 3. The model serving environment consists of OPT-6.7B, OPT-1.3B, and OPT-125M replicated across 4 GPUs. The system serves HellaSwag, COPA, PIQA, and OpenBookQA.

methodology to schedule all requests by deadline, priority, and various other application-specific constraints, as well as an option to provide smaller models at a small cost of output fidelity.

We use smaller model sizes (e.g. OPT-125M and OPT-6.7B) along with effective request stream scheduling during periods of high request load so users can achieve their desired application latency requirements while receiving acceptable quality from the LLM service. While the QUIC network protocol does not inherently support stream scheduling, the protocol packet framer can be modified to fit various scheduling paradigms. First, we choose an appropriate network stream scheduling algorithm, given user application requirements. Then, we leverage DQN, a deep Q-learning algorithm, to train a policy that determines what model to route a query to, given the task and system load. During periods of low arrival rates, we expect that a computationally efficient scheduling algorithm, such as RR or FCFS, will be able to adequately serve our requirements and we predict that our policy will route to the largest LLM with no quality degradation. As this is the expected environment most of the time, our system should rarely degrade quality. We optimize for tail request loads by scheduling streams to meet application requirements (using Earliest Deadline First (EDF), Absolute Priority (ABS), and novel application-specific MultiLevel (ML) scheduling procedures) and routing queries to small models.

Overall, the system architecture can be modeled in Figure 2. Client LLM queries reach a group of front-end servers over any network protocol or medium. The front-end servers, which can either be at the edge or in a cloud-like environment, batch requests and initiate request-response streams to the model-serving back-end server. A detailed view of the model-serving environment is shown in Figure 3. In LLM model inference, increased batch sizes result in increased model throughput, providing benefits to batch several independent requests together (Kwon et al., 2023). The batching process smooths out requests and provides consistent behavior on top of which scheduling can be introduced. The front-end servers utilize an active QUIC connection to the back-end, initiating a bi-directional QUIC stream for each request/class of requests. The request and response traffic are assigned to a stream, each which is scheduled according to its application traffic class. Each request is consumed by the model serving environment, which uses the DQN agent to select a model along with a GPU to generate an output. The RL agent analyzes the query and the load at each of the models and GPUs and determines the best model to route the query in order to maximize quality and meet latency deadlines. The model response is sent back to the corresponding front-end via the same QUIC stream, scheduled with the same semantics. Finally, these responses are delivered back to the initiating clients, completing the flow.

We do not evaluate the setting where there are multiple model-serving environments, since our work focuses on optimizing network utility within a single endpoint-to-endpoint connection and GPU utility within a single machine; however, the proposed architecture will scale accordingly. In the above scenario, the front-end servers can each select among a group of back-end servers, employing a load-balancing framework to evenly spread the request load among the group. Since QUIC streams are bi-directional, the model-serving environment simply utilizes the same incoming stream to send a response back to the appropriate front-end server.

4. Implementation

4.1. QUIC Schedulers

The open-source QUIC implementation we decided to use is `quic-go`. `quic-go` (Clemente & Seemann, 2023) is one of the original canonical implementations that is up-to-date with all current QUIC and HTTP/3 RFCs. It is actively maintained, has shown to be one of the best-performing open source implementations (Crochet et al., 2021), and is the most well-implemented (Seemann, 2023).

The default scheduling procedure in `quic-go` is Round Robin (RR). It cycles through each active stream ("active" is defined as a stream with pending data to be sent), filling up a fixed-size packet with the current stream's data before moving to the next stream and doing the same.

We extend the default RR scheduler with the following four scheduling algorithms:

1. First-Come-First-Serve (FCFS): Streams are processed in order of their arrival, with all of the data of the

first stream sent before processing the data of the next stream. We use this as a benchmark, similar to RR.

2. Earliest Deadline First (EDF): Streams are each assigned a deadline by the application and are processed in chronological order of their deadline. If a stream's deadline has passed before it has finished sending all its data, it is discarded and removed from the active stream queue.

3. Absolute Priority (ABS): Streams are each assigned a priority level by the application and are processed in order of priority (highest to lowest), with streams of the same priority served FCFS.

4. MultiLevel (ML): Streams are bifurcated into a lower and higher priority level. The higher priority streams are always served before the lower priority ones. The high priority streams are each assigned a deadline by the application and are served on an EDF basis whereas the lower level streams are served in a fair round robin manner.

In addition, we have built a scheduling abstraction layer that provides applications the functionality to select which of the stream schedulers to utilize, along with an easy way to pass in necessary information for each stream (priority for ABS, deadline for EDF and ML, etc) at stream-creation time. In our experimentation, we have enforced the same scheduler at both the client and server ends (front-end server and back-end server, respectively) but this is not necessary - connection endpoints may inter-operate between different scheduling schemes without any problem.

Stream management in `quic-go` is implemented by the `framer` interface. The interface contains a *streamQueue*, which keeps track of a list of active streams. The framer interacts with scheduling in two main ways: first when an active stream is created and added to the *streamQueue*, and second when the active stream queue is processed to send packets. In default RR, a stream gets added to the *streamQueue* at the end of the queue. While there are active streams in the *streamQueue*, the `framer` pops a stream from the front of the queue and fills up-to a packet of data from the popped stream. If the stream still has data to send, the stream is appended to the back of the queue; if not, it is removed.

First, we modified the behavior when a stream is initially added to the *streamQueue*. In ABS, the stream is added to the *streamQueue* in descending priority order (i.e. streams with the highest priority values are added to the front of the queue). Similarly, in EDF, the stream is added to the *streamQueue* in ascending deadline order (i.e. streams with the earliest deadlines are added to the front of the queue). In EDF, an additional check is performed to verify that the stream has not already missed its deadline (this check can be omitted for soft-deadline use cases). In MultiLevel (ML), we created an additional *levelTwoStreamQueue*, creating a lower priority level. Higher priority streams are added to the main *streamQueue* in ascending deadline order (as in EDF), and lower priority streams are appended to the *levelTwoStreamQueue* (as in RR). For FCFS, we append to the end of the *streamQueue*.

Next, we modify the behavior when the active *streamQueue* is being processed by the `framer` to create packets. In FCFS, EDF, and ABS, the stream we are currently processing is either the one that "came first", has the "earliest deadline", or has the "highest priority", respectively. Consequently, it remains at the front of the queue if it still has data to be sent. If the stream has no more data to send, it is removed from the queue. EDF does an additional check that discards the stream if its deadline has passed (as mentioned before, this can be omitted). The MultiLevel (ML) processes the primary *streamQueue* while it has active streams, utilizing the EDF approach above. If the primary *streamQueue* has no more active streams, the secondary *levelTwoStreamQueue* gets processed, utilizing the same behavior as RR.

4.2. RL Router Agent

We train our router's policy, represented by a 2-layer MLP with hidden size 256, using the DQN algorithm. To prevent over-estimation of Q-values we employ Double Q-learning and use a target network that gets updated every 500 iterations. We use a discount rate of 0.99 and a learning rate of 0.0001. For exploration, we use an epsilon-greedy strategy. We performed minimal hyper-parameter tuning and use this for all trained policies. All models are served using vLLM (Kwon et al., 2023), which is a state-of-the-art inference serving system. The state that the agent sees is the batch size at each model in the system, the request's task, and an estimate of the request rate. The action is the selected model, and the reward is the model's accuracy if the latency requirement is met and zero otherwise.

4.3. System Implementation

For the frontend and backend servers, we provisioned two GCP VM instances running Ubuntu 22.04 with 8 vCPUs. In order to simulate 3G network conditions (mimicking edge placement of the frontend server), we utilized the *tc* Linux kernel command, which allows a user to adjust packet drop rate, packet latency, and connection bandwidth on any specified network interface. We utilize socket IPC to forward requests to-and-from the QUIC and the model-serving runner. Apart from modifying network interface characteristics through *tc*, the network topology between the front-end and back-end servers was hidden, as is commonly the case in

cloud environments.

5. Evaluation

5.1. RL Microbenchmarks

Prior work on model serving (Li et al., 2023; Zhang et al., 2023; Gujarati et al., 2020) uses Microsoft's Azure Function (MAF) traces (Shahrad et al., 2020; Zhang et al., 2021) to model behavior of clients in a serving system. The MAF1 trace (Shahrad et al., 2020) consists of stable request periods at a fixed arrival rate before the arrival rate changes. On the other hand, the MAF2 trace (Zhang et al., 2021) has much more unpredictable client behavior and the arrival rates rapidly change. Based on these observations, we evaluate our system on three types of synthetic workloads that capture a wide range of client behavior. The first represents a stable workload in which client requests arrive in the system as a Poisson Process with a fixed rate for a set period of time. The second workload represents one in which the arrival rate of requests rapidly switches due to an underlying stochastic process that controls the arrival rate and its duration.

5.1.1. Environment

To evaluate our routing policy, we consider a serving system with 4 GPUs. Each GPU contains an instance of OPT-125M, OPT-1.3B, and OPT-6.7B, as depicted in Figure 3. When the router chooses a model size for the request, we automatically load balance by sending to the replica with the smallest batch size for the model. We set the latency guarantee to be 40 milliseconds/token. Additionally we use zero-shot HellaSwag (Zellers et al., 2019), COPA (Roemmele et al., 2011), PIQA (Bisk et al., 2020), and OpenBookQA (Mihaylov et al., 2018) as the four tasks in the system. We use each model's average accuracy on each task as a measure of its quality. For each task we normalize the accuracy of each model to OPT-6.7B's accuracy to get the rewards shown in Table 1. We pick tasks uniformly at random. We train the policy for 1.2 million iterations using hard deadlines, a uniform task distribution, and randomly chosen arrival rates.

Table 1. Rewards for tasks served in the system.

Task	OPT-125M	OPT-1.3B	OPT-6.7B
HellaSwag	0.45	0.78	1.00
COPA	0.80	0.95	1.00
PIQA	0.82	0.96	1.00
OpenBookQA	0.70	0.94	1.00

5.1.2. Stable Workload

For the stable workload, we vary the arrival rate of the Poisson Process from 0.25 to 48 requests per second and server for 40 seconds at each arrival rate before resetting and going to the next arrival rate. We show the results with hard deadlines in Figure 4. In the hard deadline setting, a client request's reward is zero if the policy does not pick an action that returns a response to the client within the latency requirements. As baselines, we show the performance when only serving to OPT-6.7B, only serving to OPT-1.3B, only serving to OPT-125M.

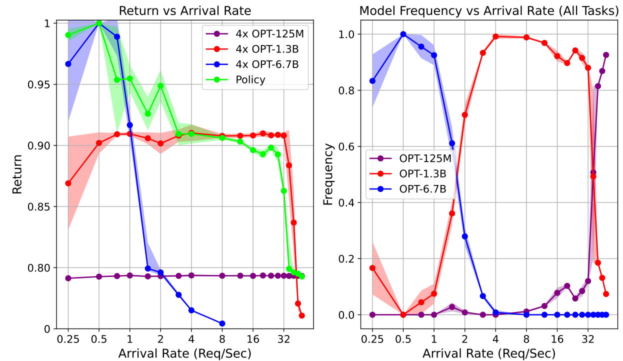

Figure 4. The left figure shows the performance on the environment with hard deadlines. The right figure shows the distribution of model selection from the policy.

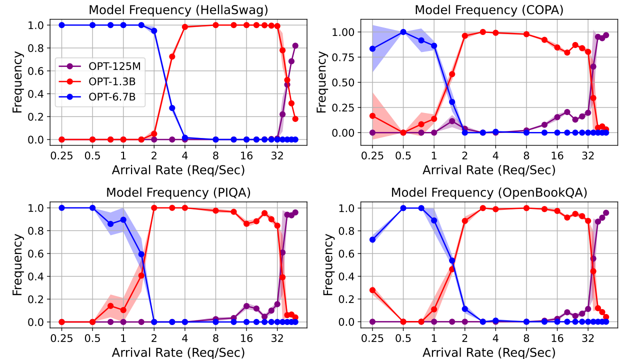

Figure 5. Model selection frequency for each individual task with hard deadlines.

As Figure 4 shows, in typical systems that serve all requests to OPT-6.7B, the performance is near the peak possible performance at low arrival rates. However, once the arrival rate increases past a threshold (2 requests per second), many latency deadlines are missed and client utility sharply declines. While OPT-1.3B can serve requests at much higher arrival rates, it's quality cannot match OPT-6.7B even when the arrival rate is low. Additionally, there is also a point at which OPT-1.3B cannot keep up with client requests. Serving only with OPT-125M leads to significant performance

degradation at all but extremely high arrival rates.

In contrast, the policy dynamically adjusts which model to send requests to. When the arrival rate is low, the policy primarily sends to OPT-6.7B and achieves similar performance. However, as the arrival rate increases, the policy correctly returns to route more requests to OPT-1.3B and eventually even OPT-125M at the extreme end. Therefore the policy allows the system to remain available for over 10x faster arrival rates than just using OPT-6.7B while still providing equal quality to OPT-6.7B at low arrival rates. Furthermore we notice that there are regions where the policy even performs better than just taking the maximum of each of the baseline's curves in Figure 4 as it is able to multiplex between models at a given arrival rate.

We now examine how the routing varies for different tasks, as shown in Figure 5. We see that the policy sends HellaSwag tasks to OPT-6.7B much more often than the other three tasks. Taking a look at Table 1, we see that OPT-125M and OPT-1.3B have a significant quality gap compared to OPT-6.7B for HellaSwag. This quality gap is much larger than the gap between models on COPA, PIQA, and OpenBookQA. Therefore the policy appropriately learns to prioritize sending HellaSwag to the large model. Furthermore, when the arrival rate is very high, HellaSwag is sent to OPT-1.3B more often than the other three tasks, for the same reason as above. Thus the router learns a complex relationship not only depending on the task's quality across models in isolation, but with respect to the quality of other tasks in the system and their distribution.

5.1.3. Unpredictable Workload

We evaluate on an unpredictable workload with large bursts, as mentioned in subsection 5.1. The first unpredictable workload, we randomly vary the arrival rate and the number of requests served at that arrival rate before switching to the next arrival rate.

Figure 6 shows the performance of the routing policy as well as the baselines, in addition to the changing arrival rate. We show both the running average of performance across all served requests and the running average of the performance across the last 20 requests. The serving system that only uses OPT-6.7B fails to meet latency deadlines during many of the bursts and thus its utility to the user is highly variable. Even though OPT-6.7B's windowed average has many averages near 1, the policy is in fact able to achieve more of these peaks. We quantify this in Table 2.

As shown in Table 2, OPT-6.7B is able to achieve more windowed averages with the top average of 1 compared to the policy. However, when analyzing the number of windows which meet high utility thresholds such as 0.99, 0.98, 0.96, and 0.94, the policy achieves more such windows than OPT-6.7B and OPT-1.3B. For example, it achieves $1.53\times$ more windows at 99% quality, $2.32\times$ more windows at 98% quality, and $4.11\times$ more windows at 96% quality compared to OPT-6.7B. Additionally, it achieves 94% of peak quality $2.75\times$ more often than OPT-6.7B and $11.18\times$ more often than OPT-1.3B. This shows that the policy is able to correctly balance between OPT-6.7B, OPT-1.3B, and OPT-125M in the same window, even while faced with an unpredictable workload.

Figure 6. Running total and windowed average over the last 20 requests of performance on the unpredictable workload. The arrival rate at each step is also shown.

Table 2. Number of request windows of size 20 that meet average quality thresholds on the first unpredictable workload.

THRESHOLD	POLICY	OPT-6.7B	OPT-1.3B
$= 1.00$	142	**307**	0
≥ 0.99	**470**	307	0
≥ 0.98	**713**	307	0
≥ 0.96	**1264**	307	0
≥ 0.94	**1723**	625	154

5.2. QUIC Scheduler Microbenchmarks

5.2.1. Experimental Setup

To evaluate our implemented schedulers, we set up a network test between the frontend and backend VM instances. As mentioned previously, we utilized the *tc* Linux kernel command to emulate various 3G network characteristics. We ran QUIC and TLS/TCP clients on the frontend VM, which initiated streams/connections to the QUIC and TLS/TCP servers on the backend instance. The server echoes all the bytes it receives back to the client leveraging the bidirectional streams of QUIC or the bidirectional con-

nections in TLS/TCP. We timed the round-trip total, starting when the client first establishes a connection/stream and ending when the client receives the final echoed byte from the server. We also evaluate the completion rates of each scheduler, based on the final round trip latency measurements.

We benchmark the newly implemented EDF, ML, and ABS schedulers against the baseline RR QUIC scheduling implementation; we also utilize the new FCFS scheduler as a baseline representing other implementations' default scheduler. Moreover, we compare our modified schedulers against TLS/TCP, since the HTTP-2/TLS/TCP stack is the industry standard protocol used in open-source model-serving applications. For TLS/TCP, we imitated stream multiplexing behavior by opening the same number of TCP connections between the client and server as we did streams in the QUIC tests. However, since we also wanted to take into account the overhead of opening many TCP connections and provide a standard of comparison against QUIC's singular connection, we also tested a 1-connection TLS/TCP setup. We test against TLS in conjunction with TCP since QUIC embeds the TLS protocol by default to provide secure transport (Iyengar & Thomson, 2021), and we wanted to provide an equal standard of comparison considering the increased computational overhead.

For our tests, we established 3 classes of traffic, representing specific LLM usage patterns. The first class of traffic is what we label "Real-Time Short" traffic, consisting of 8 KB long streams with a 1-second deadline. This traffic represents common LLM usage patterns, to be served with strict latency requirements optimizing interactive user experiences. Despite each LLM utilizing its own tokenizing methodology, a commonly quoted amount for the number of characters in a token is somewhere between 4-5 (Raf, 2023) (Kadous, 2023). 2000 tokens is well within the context window sizes of most small-sized models (OpenAI, 2023) and represents the common usage pattern of supplying a model with background contextual information followed by a question based on the context. Since most system character encodings represent each character with a byte of data, 2000 tokens is approximately 8 KB; as such, we have selected it as the size for Short Traffic. In EDF this traffic class is assigned a 1-second deadline, in ABS it is assigned the highest priority level, and in ML it is assigned the higher priority queue. We made it our primary objective to schedule streams to achieve as low of a latency and as high of a completion rate as possible for this traffic class.

The second class of traffic is what we label "Free-Tier Short" traffic, consisting of the same 8 KB request streams as the above Real-Time Short, but representing traffic with more lenient deadlines and QoS requirements - perhaps "free-tier" traffic for LLM providers. In EDF this traffic class is assigned a 2-second deadline, in ABS it is assigned the lower priority level, and in ML it is assigned the lower priority queue.

Finally, we label the third class of traffic as "Long Output" traffic. This traffic class consists of 102.4 KB long streams with a 7-second deadline, representing long-input, long-output corpora synthesizing tasks. 25,000 tokens is on the higher end of context window sizes but is attainable by today's state-of-the-art advanced LLMs (OpenAI, 2023) (Anthropic, 2023). In EDF this traffic class is assigned a 7-second deadline, in ABS it is assigned the lower priority level, and in ML it is assigned the lower priority queue.

We simulate the 3G network via *tc*, dialing in 5 Mbps of bandwidth, 0.5% packet loss, and 50 ms of added packet latency (Chan & Ramjee, 2002). The client sends 90 requests to the server, randomly selecting a traffic class (with uniform probability), and recording the time from when the stream/connection is first established, to when last byte of the request is echo-ed back from the server. The latency of each request is recorded, and a completion rate is calculated based on the request class deadline.

5.2.2. Discussion

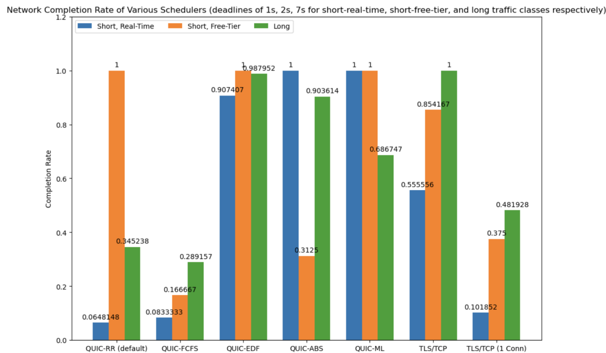

Figure 7. Network completion rates of the 3 traffic classes using various scheduling algorithms and network protocols. Short, Real-Time tasks with 1-second deadline; Short, Lower-Priority tasks with 2-second deadline; Long Context Window, Lower-Priority tasks with 7-second deadline.

From the box plot of latency distributions, we see that QUIC-RR and TLS/TCP (n-connections) performed similarly. Intriguingly, the performances of the two protocols are quite evenly matched despite the on-paper advantages that QUIC has. We hypothesize several possible reasons - increased overhead of a large number of streams multiplexing a single QUIC connection, TCP integration and optimization for Golang (TCP is part of the standard language library whereas quic-go is independent), UDP kernel buffer size restrictions, and QUIC's strict maximum packet size limits - however, we leave investigation out of scope for this work.

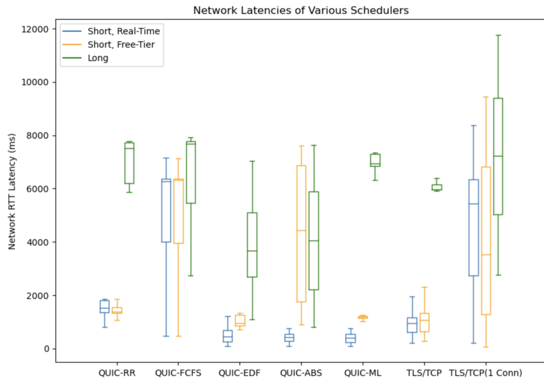

Figure 8. Network request latencies of the 3 traffic classes using various scheduling algorithms and network protocols.

When analyzing completion rates, we see that both QUIC-RR and TLS/TCP struggle to achieve high completion rates for high-priority traffic, which is our most important metric. TLS/TCP 1-connection latency results underscore the head-of-line blocking issues present in the protocol with exceptionally high tail latencies for all 3 traffic classes. We see a similar issue in QUIC-FCFS, where long-output streams block progress for both high-priority and low-priority short streams.

Given the primary metric used in LLM model-serving tasks is per-token latency and deadline SLOs, we decided to first build an EDF scheduler. From the boxplots, we can see that it outperforms QUIC-RR in latency for all traffic classes. Furthermore, QUIC-EDF achieves high completion rates across all 3 traffic classes, reaching 100% for short/low-priority. However, since short/low-priority stream deadlines can occur before short/high-priority stream deadlines (due to random selection and order of stream creation), high-priority traffic can be preempted and thus miss its deadline. As a result, we see sub-100% completion rates for high-priority traffic prompting us to build two other schedulers aimed to prevent the preemption of high-priority traffic.

Next, we implemented an absolute priority scheduler and evaluated its performance. Since streams would be served on a strict priority basis, preemption of high-priority traffic would be avoided. As a result, we observe a 100% completion rate and near-optimal latency of high-priority traffic, accomplishing our primary scheduling metric. However, we note a huge drop-off in the performance of the low-priority/short traffic class, with especially degraded tail latencies. Since the priority scheduler serves each priority class on a first-come-first-serve basis, we see that low-priority/short traffic gets blocked behind low-priority/long traffic. This could be solved by assigning long-output traffic a lower priority; however, we reasoned that this would not match the intended semantics of the traffic classes.

In response, we created a novel multi-level scheduling scheme, to address both preemption and head-of-line blocking issues seen in previous schedulers. The multi-level scheduler addressed the high-priority preemption issues by placing high-priority traffic in its own EDF higher-level scheduling queue while also addressing low-priority/short traffic blocking issues by utilizing a round-robin approach for the lower-priority queue. Overall, we deem the ML scheduler to perform best since it meets the primary criteria achieving 100% completion rate and optimal latencies for high-priority traffic while also serving both classes of low-priority traffic with strong guarantees. We also observe the most consistent behavior with the ML scheduler, evidenced by the significantly lower latency variance in the boxplot.

5.3. Macrobenchmark

We now evaluate the performance of our network scheduling in combination with the learned router agent. Network scheduling and dynamic model selection can be applied in a variety of settings. However, for our evaluation, we focus on LLM applications that serve three classes of requests:

1. High priority traffic with small input sizes
2. Low priority traffic with small input sizes
3. Low priority traffic with large input sizes

As described in the above microbenchmark sections, an application may need to serve small input sizes if it is answering questions such as ”How to seal wood?” The application may need to serve queries with large input sizes to answer questions such as ”Summarize the following document about sealing wood: ...”. We evaluate various network schedulers and serving mechanisms on this workload, using both stable and unpredictable arrival patterns.

The following are our metrics of success to test during evaluation:

- Meet user-defined latency deadlines and QoS requirements, achieving high request completion rates.
- Serve highest possible model quality given resource contention
- Handle both stable and unpredictable client requests patterns across a wide range of client request rates.

We show the average performance on both the stable and first unpredictable workload from subsection 5.1 in Table 3 and Table 4. Our performance score combines both model quality and latency. If request's latency deadline is met, the performance score for the request is the selected model's accuracy. Otherwise, if the latency deadline is not met,

the performance is zero. Similarly, we serve HellaSwag, COPA, PIQA, and OpenBookQA. We denote OpenBookQA to be high priority traffic. The policy recognizes this by up-weighting the reward for serving OpenBookQA at high quality. We train the policy for 1.1 million iterations using OPT-2.7B as the larger model and OPT-1.3B as the smaller model. We see that the policy outperforms static serving baselines on the stable workload, but only outperforms OPT-2.7B on the unpredictable workload. We see that OPT-1.3B outperforms the policy in the unpredictable workload. We believe this is because the policy does not take into account the network latency when making its decision, although it takes into account the priority of the task. We leave this as future work.

Table 3. Performance on the stable workload.

SCHEDULER	OPT-2.7B	OPT-1.3B	POLICY
QUIC-RR	0.20	0.25	**0.42**
QUIC-FCFS	0.19	0.24	**0.36**
QUIC-EDF	0.21	0.37	**0.52**
QUIC-ABS	0.20	0.35	**0.49**
QUIC-ML	0.21	0.29	**0.45**
TLS/TCP	0.20	0.24	**0.41**
TLS/TCP (1 CONN)	0.18	0.22	**0.31**

Table 4. Performance on the unpredictable workload.

SCHEDULER	OPT-2.7B	OPT-1.3B	POLICY
QUIC-RR	0.19	**0.54**	0.42
QUIC-FCFS	0.17	**0.53**	0.36
QUIC-EDF	0.19	**0.64**	0.48
QUIC-ABS	0.18	**0.62**	0.46
QUIC-ML	0.19	**0.57**	0.44
TLS/TCP	0.18	**0.54**	0.41
TLS/TCP (1 CONN)	0.16	**0.51**	0.35

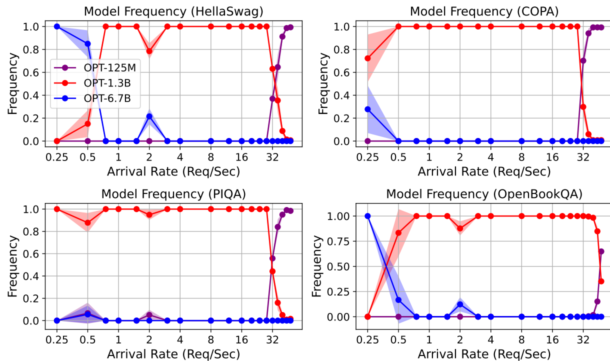

Figure 9. Model selection frequency for each individual task with OpenBookQA prioritized.

We also investigate the quality of OpenBookQA, which is the prioritized task in our system. As shown in Figure 9, OpenBookQA and HellaSwag are sent to OPT-6.7B significantly more often than COPA and PIQA. This is because HellaSwag is a hard task whose quality benefits significantly from OPT-6.7B, and OpenBookQA is a prioritized task in the system. Additionally, at high arrival rates, the system favors sending OpenBookQA to OPT-1.3B while it sends the other tasks to the smaller OPT-125M.

6. Future Work

There are a number of future directions to be explored to improve the router agent. When running multiple models on a GPU, there are scheduling decisions on the systems side that need to be made to determine how models share compute resources to further help meet latency deadlines. Additionally, it will be interesting to see extensions to the work that use embeddings or other ways to expand the state in order to capture further information about a client's request and the state of the system. On the network scheduling end, further exploration into the throughput tradeoffs between QUIC and TLS/TCP is necessary, as our initial benchmarks showed similar results between the two baselines. We also believe that the overall system performance can be improved by having the router agent take into account the network latency for a request, rather than just its priority. In addition, we can also assess the performance of multiple model-serving environments. We envision an environment where front-end servers can each select among a group of back-end servers, employing a load-balancing framework to evenly spread the request load among the group.

7. Conclusion

In this work, we have presented an end-to-end quality-of-service aware model serving system. We utilized a Q-learning-based routing agent to dynamically choose between models of different latency and quality during periods of high request load, maintaining availability and serving at the highest possible quality. We have also designed and implemented a series of stream scheduling algorithms for the QUIC network protocol. Wee identified the SLO requirements of various classes of LLM requests and utilized both network scheduling as well as dynamic model selection to meet application requirements. We have trained the Q-learning agent to optimize latency and model serving quality of higher priority classes while achieving fairness and best-effort response quality for lower priority requests. We have benchmarked the QUIC network schedulers and have shown that the EDF and Multilevel schedulers perform best, achieving the highest completion rates and lowest network latencies for high-priority traffic while maintaining high completion rates for lower-priority traffic as well. Overall,

the network schedulers have shown increased performance over the baseline QUIC protocol, as well as TLS/TCP. We benchmark our end-to-end system against today's model-serving standard and demonstrate the ability to not only cater to QoS differentiation between requests but also outperform current standards in per-token latency and model-output quality.

References

Agarwal, M., Qureshi, A., Sardana, N., and Li, L. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023. Accessed: 12/14/2023.

Anthropic. Introducing 100k context windows, 2023. URL https://www.anthropic.com/index/100k-context-windows. Accessed: 12/14/2023.

Belshe, M., Peon, R., and Thomson, M. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

Bishop, M. HTTP/3. RFC 9114, June 2022.

Bisk, Y., Zellers, R., Bras, R. L., Gao, J., and Choi, Y. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

Bornstein, M. and Radovanovic, R. Emerging architectures for llm applications. https://a16z.com/emerging-architectures-for-llm-applications/, 2023. Accessed: 12/14/2023.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Cettolo, M., Federico, M., Bentivogli, L., Jan, N., Sebastian, S., Katsuitho, S., Koichiro, Y., and Christian, F. Overview of the iwslt 2017 evaluation campaign. In *Proceedings of the 14th International Workshop on Spoken Language Translation*, pp. 2–14, 2017.

Chan, M. and Ramjee, R. Tcp/ip performance over 3g wireless links with rate and delay variation. volume 11, pp. 71–82, 09 2002. doi: 10.1007/s11276-004-4748-7.

Chiariotti, F., Deshpande, A. A., Giordani, M., Antonakoglou, K., Mahmoodi, T., and Zanella, A. Quicest: A quic-enabled scheduling and transmission scheme to maximize voi with correlated data flows. *IEEE Communications Magazine*, 59(4):30–36, 2021. doi: 10.1109/MCOM.001.2000876.

Clemente, L. and Seemann, M. quic-go: A quic implementation in pure go, 2023. URL https://github.com/quic-go/quic-go. Accessed: 12/14/2023.

Crochet, C., Rousseaux, T., Piraux, M., Sambon, J.-F., and Legay, A. Verifying quic implementations using ivy. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, EPIQ '21, pp. 35–41, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391351. doi: 10.1145/3488660.3493803.

Cui, C., Lu, Y., Li, S., Li, J., and Ruan, Z. Dash+: Download multiple video segments with stream multiplexing of quic. In *2022 Tenth International Conference on Advanced Cloud and Big Data (CBD)*, pp. 66–72, 2022. doi: 10.1109/CBD58033.2022.00021.

Fernández, F., Zverev, M., Diez, L., Juárez, J. R., Brunstrom, A., and Agüero, R. Flexible priority-based stream schedulers in quic. In *Proceedings of the Int'l ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*, PE-WASUN '23, pp. 91–98, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703706. doi: 10.1145/3616394.3618267.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. Teaching machines to read and comprehend. *Advances in neural information processing systems*, 28, 2015.

Iyengar, J. and Swett, I. QUIC Loss Detection and Congestion Control. RFC 9002, May 2021.

Iyengar, J. and Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

Kadous, W. Numbers every llm developer should know, 2023. URL https://www.anyscale.com/blog/num-every-llm-developer-should-know. Accessed: 12/14/2023.

Kim, S., Mangalam, K., Malik, J., Mahoney, M. W., Gholami, A., and Keutzer, K. Big little transformer decoder. *arXiv preprint arXiv:2302.07863*, 2023.

Kutter, M. and Jaeger, B. Comparison of different quic implementations. In *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM)*, 2022.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving

with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C. B., Shi, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J. C., Roskind, J., Kulik, J., Westin, P. G., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., and Chang, W.-T. The quic transport protocol: Design and internet-scale deployment. 2017.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *arXiv preprint arXiv:2302.11665*, 2023.

Mihaylov, T., Clark, P., Khot, T., and Sabharwal, A. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*, 2018.

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

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

OpenAI. Openai models, 2023. URL https://platform.openai.com/docs/models. Accessed: 12/14/2023.

Pauly, T., Kinnear, E., and Schinazi, D. An Unreliable Datagram Extension to QUIC. RFC 9221, March 2022.

Raf. What are tokens and how to count them? https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them, 2023. Accessed: 12/14/2023.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

RayTeam. Ray serve: Scalable and programmable serving, 2023. URL https://docs.ray.io/en/latest/serve/index.html. Accessed: 12/14/2023.

Roemmele, M., Bejan, C. A., and Gordon, A. S. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI Spring Symposium Series*, 2011.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Seemann, M. Quic interop runner, 2023. URL https://interop.seemann.io/. Accessed: 12/14/2023.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pp. 205–218, 2020.

Thomson, M. and Benfield, C. HTTP/2. RFC 9113, June 2022.

Thomson, M. and Turner, S. Using TLS to Secure QUIC. RFC 9001, May 2021.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

vLLM Team. vllm: Quickstart, 2023. URL https://docs.vllm.ai/en/latest/getting_started/quickstart.html. Accessed: 12/14/2023.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Zhang, H., Tang, Y., Khandelwal, A., and Stoica, I. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 787–808, 2023.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhang, Y., Goiri, Í., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C., and Bianchini, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 724–739, 2021.

Chapter 2

Scheduling in MASQUE Proxies

In this second chapter, we investigate another application of stream scheduling in QUIC: proxying. Multiplexed Application Substrate over QUIC Encryption (MASQUE) is a new framework that enables QUIC-based tunneling to a proxy. Proxies are widely deployed in the internet today; however, providing security and privacy guarantees to user traffic while maintaining application performance at scale continues to be a challenge. We utilize stream scheduling to solve performance degradations seen in MASQUE proxies when delivering data with reliable streams. Moreover, we implement the PIFO data structure in MASQUE to enhance stream scheduling performance at scale. Our results demonstrate that stream scheduling behavior has significant performance implications in MASQUE tunneling - naive schedulers not only deteriorate proxy performance by inducing packet losses and spurious retransmits but also fail to capture the priority and QoS semantics of end-to-end connections. Additionally, we show that the PIFO-based priority scheduler improves stream management overhead and latency of end-to-end priority-based MASQUE connections at scale.

2.1 Introduction

Proxies are widely used in today's internet infrastructure, assuming various forms such as VPNs, firewalls, CDNs, load-balancers, performance enhancers, and tunnels. It is estimated that a significant fraction of all end-user HTTP traffic passes through web proxies [20]. Studies have detected middleboxes in approximately 80% of observed networks [19] and have even observed an equal number of middleboxes and routers in some enterprise networks [15]. Many such reverse proxies are transparent to the end-user, and can only be detected with specific inspection techniques. Conversely, some users may wish to explicitly use forward proxies for accessing restricted web content, protecting user identity, and securing user traffic.

TCP/UDP forwarding is a rudimentary form of proxying, where the middlebox forwards the payload of received packets to a target destination using static host/port forwarding rules. Not only is this approach limiting, but it is also insecure as the proxy has full access to the unencrypted payload. SOCKS proxies provide greater flexibility than static host/port forwarding but continue to lack encryption and privacy guarantees. VPN protocols, such as IPSec and Wireguard, are secure and privacy-enabling alternatives; however, they often require administrator privilege in the network and, as such, may not always be viable. Moreover, Wireguard uses non-standard protocols which are regularly blocked by firewalls and network boundaries. HTTP Proxies use TLS to provide a secure tunneling interface and use first-class network protocols such as TCP and UDP to avoid being blocked by networks. Early HTTP/1 and HTTP/2 proxies rely on TCP, which lacks connection multiplexing and suffers from Head-of-Line blocking, as we discussed previously. Moreover, Perino et al. have shown that TLS Interception continues to be an issue in these proxies [11].

MASQUE is a network framework being standardized by an IETF working group of the same name. It enables the use of HTTP/3 and QUIC as a tunneling protocol, thereby inheriting all its benefits (fast connection establishment, no HoL blocking, flexible congestion control, built-in TLS encryption, stream abstractions, etc). MASQUE utilizes the CONNECT-UDP HTTP method (standardized in RFC 9298) to establish a connection to a proxy server and request tunneling to a target (remote) server [13]. The client sends a CONNECT-UDP HTTP/3 request to a proxy which subsequently opens a UDP socket to the target server. The client extracts the underlying QUIC transport from the HTTP/3 connection which it uses to send encapsulated QUIC datagrams. The proxy decapsulates the datagrams and forwards them to the remote server. The proxy re-encapsulates packets returning from the server before delivering them back to the client. MASQUE supports secure UDP tunneling, is implemented at the application level, and looks like normal HTTP traffic to firewalls, thus addressing the limitations of previous proxying techniques.

MASQUE is actively used by internet-scale companies such as Apple [9], Akamai [12], and Cloudflare [8] to provide privacy, CDN, and VPN services to users. Most notably, Apple's iCloud Private Relay [5] leverages MASQUE to protect user privacy on all Apple devices,

leaving no single party with knowledge of both the client's IP identity and its browsing activity. All outgoing network traffic is encrypted and reaches a series of two "relay" (MASQUE proxy) servers. The first hop server tunnels traffic from the user through to the second hop. It knows the user's IP address but cannot access its encrypted DNS request. The second hop is able to decrypt the DNS request and resolve a destination IP address but has no information about the user's original IP address since it sees the data as coming from the first hop server. The final target destination receives tunneled application payload from the second hop, with no knowledge of the original client's identity.

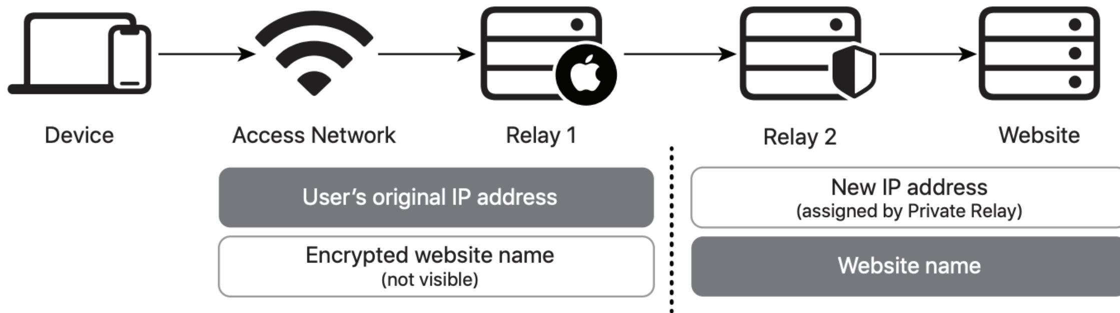

Figure 2.1: Apple's Dual-Hop Relay Architecture

Despite the widespread adoption of MASQUE in end-user applications, open challenges regarding stream scheduling and scalability of stream multiplexing exist. Section 2.2 provides details on our MASQUE proxying architecture, the role of stream multiplexing in MASQUE, and the performance and scalability challenges in MASQUE scheduling; Section 2.3 describes our MASQUE implementation, tunneled stream scheduling semantics, and PIFO data structure implementation; Section 2.4 presents our evaluation setup and results; Section 2.5 provides an outlook for future work and Section 2.6 concludes.

2.2 Background

QUIC Streams vs QUIC Datagrams

The MASQUE Working Group has released two RFCs for the MASQUE framework: RFC 9297 [14] which introduces the concept of HTTP Datagrams, and RFC 9298 [13] which details the CONNECT-UDP mechanism to proxy QUIC. HTTP Datagrams, which employ QUIC datagrams for transport instead of QUIC streams, are transmitted unreliably [10]. Consequently, packet loss does not hamper the congestion control send window. On the other hand, RFC 9297 also defines the notion of CAPSULEs, which transmit HTTP Datagrams inside a reliable stream protocol, such as standard QUIC. MASQUE suggests HTTP Datagrams over CAPSULEs, citing performance drawbacks from layered loss recovery and congestion control. When a reliable transport tunnels inside another reliable transport, like QUIC-over-QUIC, nested congestion control tends to amplify its response to congestion and packet loss, deteriorating throughput. However, Kuhlewind et al. have demonstrated that HTTP/3 CAPSULEs, utilizing QUIC streams, demonstrate better end-to-end connection latencies in lossy networks than HTTP datagrams [6]. The authors explain that the outer tunnel in stream mode repairs losses before the end-to-end connection detects it, preventing a significant reduction in congestion control throughput. The paper also points out that stream mode outperforms datagram mode as the number of simultaneous connections increases.

MASQUE Connection Architecture

While RFC 9298 provides the semantics of CONNECT-UDP requests and responses, the released MASQUE documents don't specify the actual tunneling procedures. Consequently, a proliferation of techniques using different transport modes, tunneling mechanisms, packet encapsulations, tunneling layers, end-to-end layers, and proxy abstractions have been implemented [5] [6] [8] [17] [18]. In the subsequent section, we detail a mixed architecture combining elements that best suit our chosen open-source QUIC implementation and most align with our requirements and objectives.

To first establish a MASQUE tunnel, a client must send an HTTP/3 CONNECT-UDP request to a MASQUE-aware proxy server. CONNECT-UDP is a standard HTTP request method, similar to GET or POST. Inside the request, the client specifies the destination server in the Path header field and the proxy server in the Authority header field. The proxy server accepts the request and responds to the client with a 2XX indicating success.

The proxy simultaneously opens a UDP socket to the remote server specified in the client's MASQUE request, performing DNS resolution if necessary. Since UDP is not a connection-based protocol, the proxy does not need to wait for connection establishment to respond successfully to the client, reducing RTTs and proxy startup time.

```
CONNECT HTTP/3
Method: CONNECT
Protocol: connect-udp
Scheme: https
Path: /target.destination.com/443/
Authority: proxy.server.com
```

Figure 2.2: Example HTTP/3 CONNECT-UDP Request

Upon receiving the proxy's response, the client can begin to tunnel UDP packets to the destination server. The client first extracts the HTTP/3 session's underlying QUIC connection and opens a new stream in the extracted connection. The client encapsulates handshake packets within the extracted QUIC stream. Figure 2.3 illustrates tunnel encapsulation for QUIC packets.

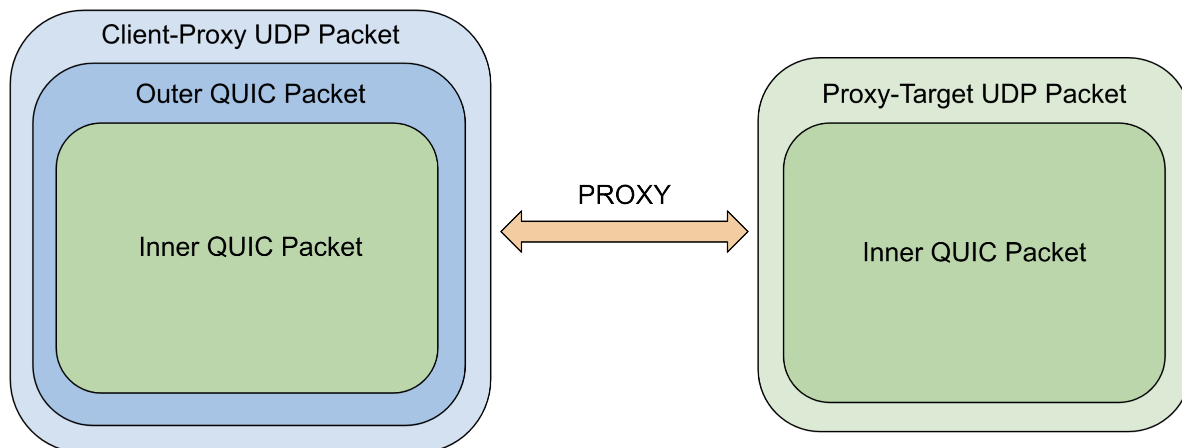

Figure 2.3: Tunnel Packet Encapsulation and Decapsulation

We refer to the connection between the client and the proxy as the "outer" QUIC connection or the "tunnel" and the connection between the client and the target as the "inner" QUIC connection or the "end-to-end" connection. It is important to note that the outer connection is encrypted using keys that the client and proxy share prior to the exchange of

HTTP messages. Therefore, the proxy must decrypt the received packets during decapsulation. However, the inner QUIC packets are secured with keys that the client and destination server share during connection establishment. As a result, the proxy is unable to read inner connection data. It treats the ciphertext as UDP payload and prepends the packets with a UDP header as they depart the proxy's socket to the target server. Packets from the destination server are re-encapsulated by removing the UDP header, encrypting the data with the outer connection keys, and prefixing the outer UDP and QUIC headers.

Upon a successful handshake with the destination server, the client is able to send data and manage streams in the inner connection. An illustration of the end-to-end tunnel connection architecture is shown in Figure 2.4.

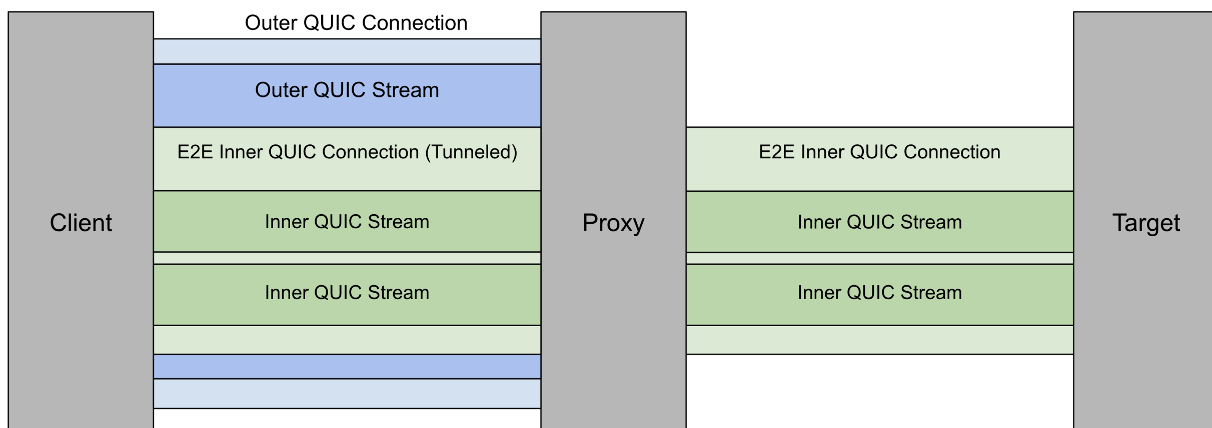

Figure 2.4: Tunnel Connection Architecture

A subtle but important note is that the target server sees the inner QUIC connection as an exchange between itself and the proxy server, based on the IP and UDP headers it receives. It is unaware of the client's existence. However, the proxy knows both the client's IP and the target server's identity; Apple's iCloud Relay introduces an extra proxy server to split this information between two parties.

This architecture provides the opportunity to re-use a single proxy connection to tunnel to several different target servers. Figure 2.5 depicts an architecture with one client, one

proxy, and several target servers. In order to establish a connection to a new target server, the client simply opens a new stream in the existing outer connection and tunnels a handshake to the new target. This greatly reduces connection setup time and facilitates scalability.

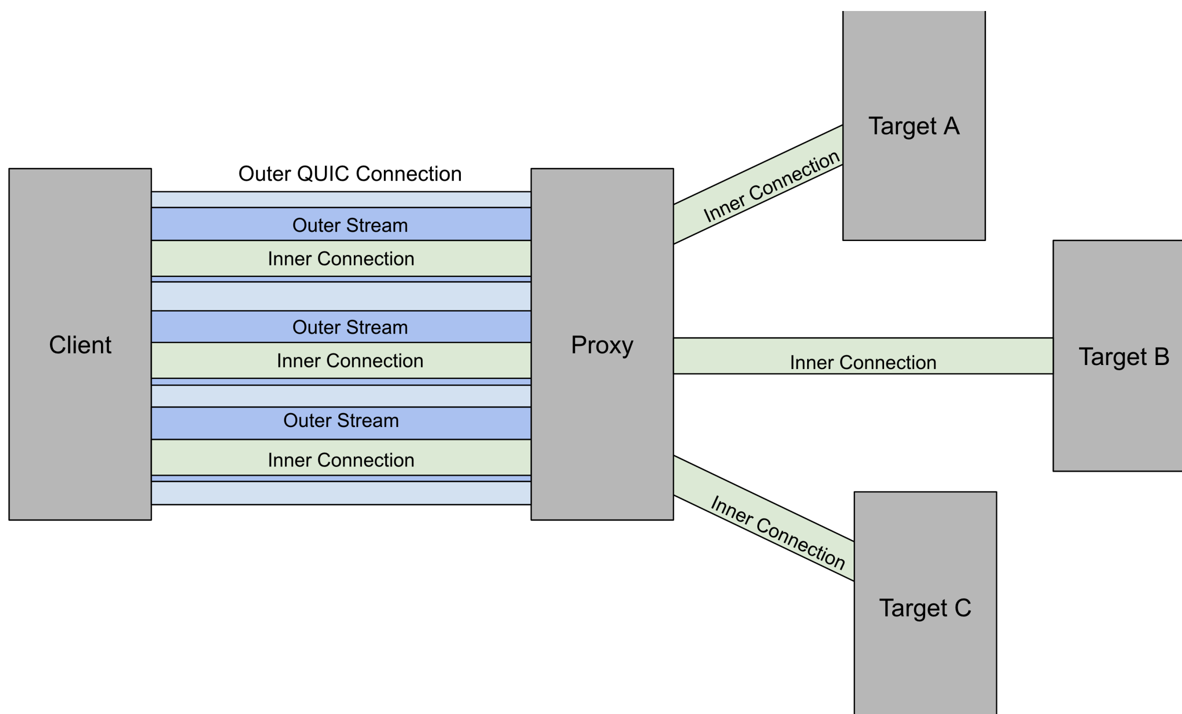

Figure 2.5: Tunnel Connection Architecture for Single Client, Single Proxy, and Multiple Targets

In a network with multiple proxies, the client has a choice on which proxy to utilize. In the event that the current proxy is congested or unavailable, or if the client moves closer to another proxy (as in the case of cellular networks), the client would benefit from switching proxies. Since QUIC connections are identified by their Connection ID and not a 5-tuple, migrating the inner connections is a simple, low-latency operation. The client opens an outer connection to the new proxy, which takes a single RTT (using QUIC's 1-RTT handshake). All inner connections are seamlessly tunneled through the new proxy, experiencing minor delays. The proxy only needs to allocate a new UDP socket to the target. To the best of our knowledge, few works have explored QUIC connection migration in the context of MASQUE proxies and we believe that this presents an exciting avenue for future work and exploration.

Stream Scheduling in MASQUE

Stream management plays a crucial role in the MASQUE architecture described above. In order for a client to simultaneously connect to multiple destinations, it must open and manage several streams between itself and the proxy. Without the required stream scheduling primitives, the system cannot capture the priority semantics between end-to-end connections. Additionally, MASQUE architectures may allow a client to request quality-of-service for certain flows; this requires the proxy to schedule the outer streams appropriately to meet service requirements [6]. Moreover, each end-to-end connection can itself contain many independent streams, creating a structure of nested streams - scheduling and priority must be respected between interleaving layers of streams. In this model, both inner and outer streams are visible to the client, outer streams are visible to the proxy, and inner streams are visible to the destination; consequently, the client is responsible for managing scheduling semantics in both the end-to-end and tunnel flows.

Kuhlewind et al. have observed performance deterioration in MASQUE proxies using stream mode with default stream scheduling behaviors. As the number of simultaneous end-to-end QUIC connections through the proxy increased, they noted a rise in packet loss. They attributed the increased packet loss to the default stream scheduling behavior in their selected QUIC implementation, aioquic [7]. Aioquic utilizes FCFS stream scheduling where only the first stream makes progress while all the other streams block. As the number of simultaneous end-to-end connections increased, each connection had to be buffered at the proxy for longer. Consequently, blocked connections experienced timeouts, leading to a significant decrease in connection throughput caused by congestion control interpreting timeouts as packet loss. Timeouts also prompted spurious retransmits, further increasing network congestion. We expect a basic scheduling substitute, like Round Robin, to address this issue.

Scaling Stream Management

As the deployment of MASQUE continues, the importance of scalability becomes increasingly apparent and demands greater attention. Pauly notes that Apple has witnessed thousands of concurrent streams in its iCloud Relay proxies, with active connection durations surpassing those previously tested [9]. Without scalable stream management, scheduling becomes a performance bottleneck. In our stream scheduling implementation outlined in the preceding chapter, the stream management overhead accounts for approximately 30% of the non-network-related task overhead. Furthermore, we observe that this overhead increases as the number of simultaneous streams also increases. Although unavoidable, improved scheduling implementations that utilize better stream management data structures can mitigate this problem.

2.3 Implementation

MASQUE

We opted to implement MASQUE into `quic-go` [4], having already used it for our stream scheduling implementation. Since MASQUE is a relatively recent protocol, the majority of open-source QUIC implementations, including `quic-go`, have yet to adopt it.

The first step in adding support for MASQUE is implementing the CONNECT-UDP HTTP method. `quic-go` already supports HTTP/3 so it was just a matter of specifying the Method field in the request. The proxy server checks for CONNECT-UDP in the Method field of the received request header and responds with a 200. The crux of the implementation is converting the HTTP/3 request-response flow into a QUIC stream and tunneling QUIC packets inside the stream. We added support for "connection hijacking" which allows us to extract the HTTP/3 connection from the request-response exchange and cast it to a QUIC connection. We can now open a stream within this connection to start tunneling our packets. In `quic-go`, it's necessary to specify the underlying transport to dial the connection (referred to as `net.Conn`). By wrapping the QUIC stream and implementing a few methods, we can fulfill the requirements of the `net.Conn` interface, thereby enabling us to tunnel the inner connection through the outer stream.

On the proxy side, we repeat the steps to extract the QUIC stream from the HTTP/3 flow and wrap it in a struct that implements `net.Conn`. We create a UDP socket to the destination server and plug in the wrapped stream. Data received by the outer QUIC stream is automatically decrypted, packetized, and forwarded to the destination server. The remote server interprets the received packets as a regular handshake coming from the client.

PIFO

To tackle the scalability challenges related to stream management, we have integrated a PIFO-based data structure into our stream multiplexer. PIFO, which stands for Push-In-First-Out, was introduced by Sivaraman et al. for scheduling packets in programmable switches [16]. It defines an abstraction enabling users to insert packets into the structure at any position (based on their relative priorities), but exclusively retrieve packets from the queue's head. Despite challenges in implementing this data structure in programmable switch hardware [1] [2], this data structure is easily realizable in software as a heap-based priority queue. We store the stream's priority along with a pointer to the stream's data in the priority queue. Since we're not operating at line rate in software, we are able to provide a pointer to the data rather than the packets themselves in the queue.

2.4 Evaluation

Setup

We repeat the implementation setup from the previous chapter, this time provisioning three GCP VM instances (for the client, proxy, and target) running Ubuntu 22.04 with 8 vCPUs. We simulated 3G network conditions using the `tc` Linux kernel command, allowing us to adjust packet drop rate, packet latency, and connection bandwidth. As is usually the case in cloud environments, the network topology between the three VMs was hidden.

MASQUE Scheduling Microbenchmark

Our first benchmark aimed to recreate the flow buffering and spurious retransmit issues faced by the MASQUE FCFS scheduler described by Kuhlewind et al. in [6]. We analyze the FCFS scheduler alongside a basic Round Robin alternative, showcasing the performance impacts of different stream scheduling behaviors in MASQUE. In our test environment, the client opens up 100 separate inner connections to the target server, tunneling through 100 corresponding outer streams. This setup emulates the one-client/one-proxy/many-target architecture that was detailed above.

Each connection tunnels 10 KB of data to the target server, which echoes the data back to the client via the tunnel. We timed the round-trip latency, starting when the client first opens a new tunnel stream and ending when the client receives the final echoed byte from the server. We also evaluate the completion rates of each scheduler, based on the final round-trip latency measurements. We dial in 5 Mbps of bandwidth, 1.5% packet loss, and 50 ms of added packet latency with the `tc` command.

Comparing the completion rates of the two schedulers in Figure 2.6, we see a marked improvement in switching our outer stream scheduler from FCFS (17% completion rate) to RR (68% completion rate) for a deadline of 8 seconds. The Round Robin scheduler faces far fewer timeouts and spurious retransmits since flows aren't buffered at the proxy without progress.

Examining the boxplot of stream round trip latencies in Figure 2.7, we observe a decrease in median RTT latency when transitioning from FCFS to RR. The first few FCFS connections finish in under 5000 ms; however, as congestion at the proxy builds, the completion times of the streams escalate rapidly. The RR scheduler is comparatively much more consistent, with more than half of all streams completing between 5500 and 7500 ms. We do note a skew in completion times towards the tail end of RR streams and hypothesize that untimely packet drops may have cut throughput significantly for later streams.

Additionally, we observe an increase in end-to-end throughput, with all 100 streams completing in approximately 15 seconds under the RR scheduler, compared to around 24 seconds

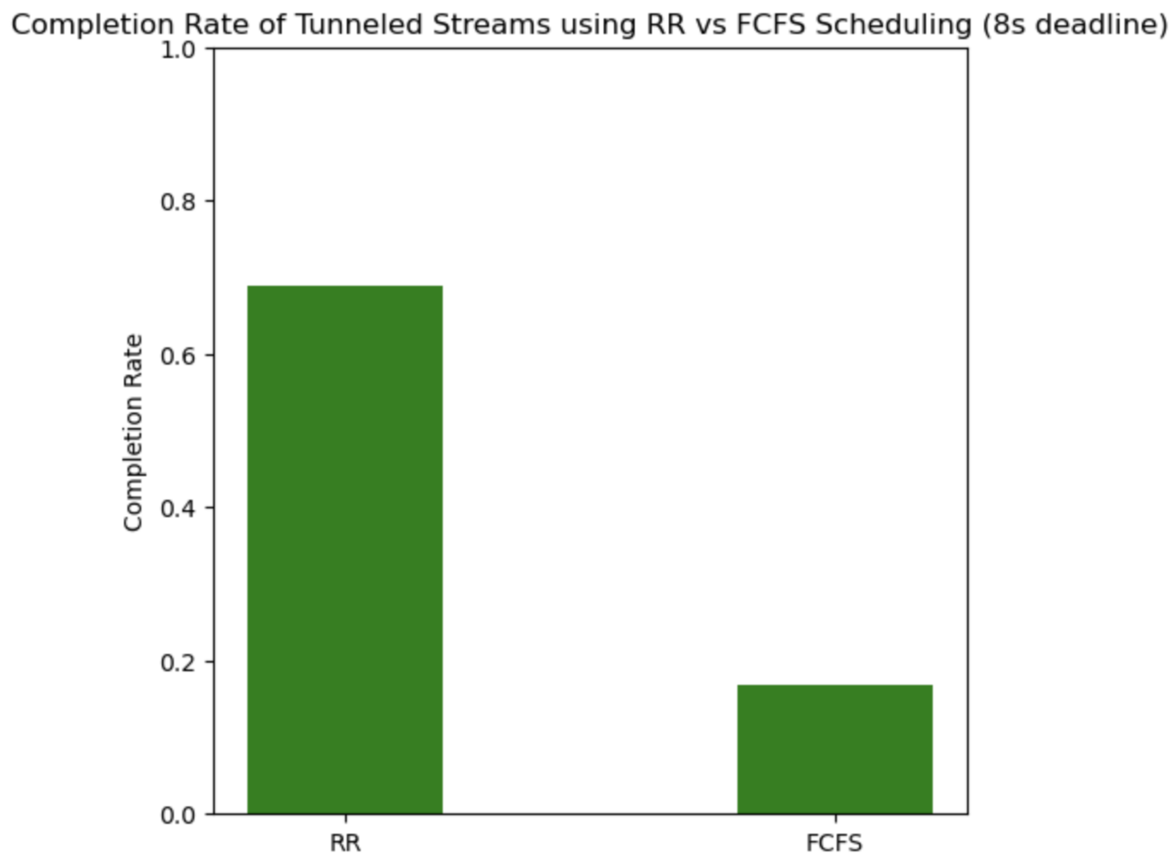

Figure 2.6: Completion Rate of Tunneled Streams using RR vs FCFS Scheduling (8s deadline)

under the FCFS scheduler (displayed in Figure 2.8). We postulate that the overall throughput of the RR scheduler would be higher if not for the tail-end streams experiencing untimely packet drops.

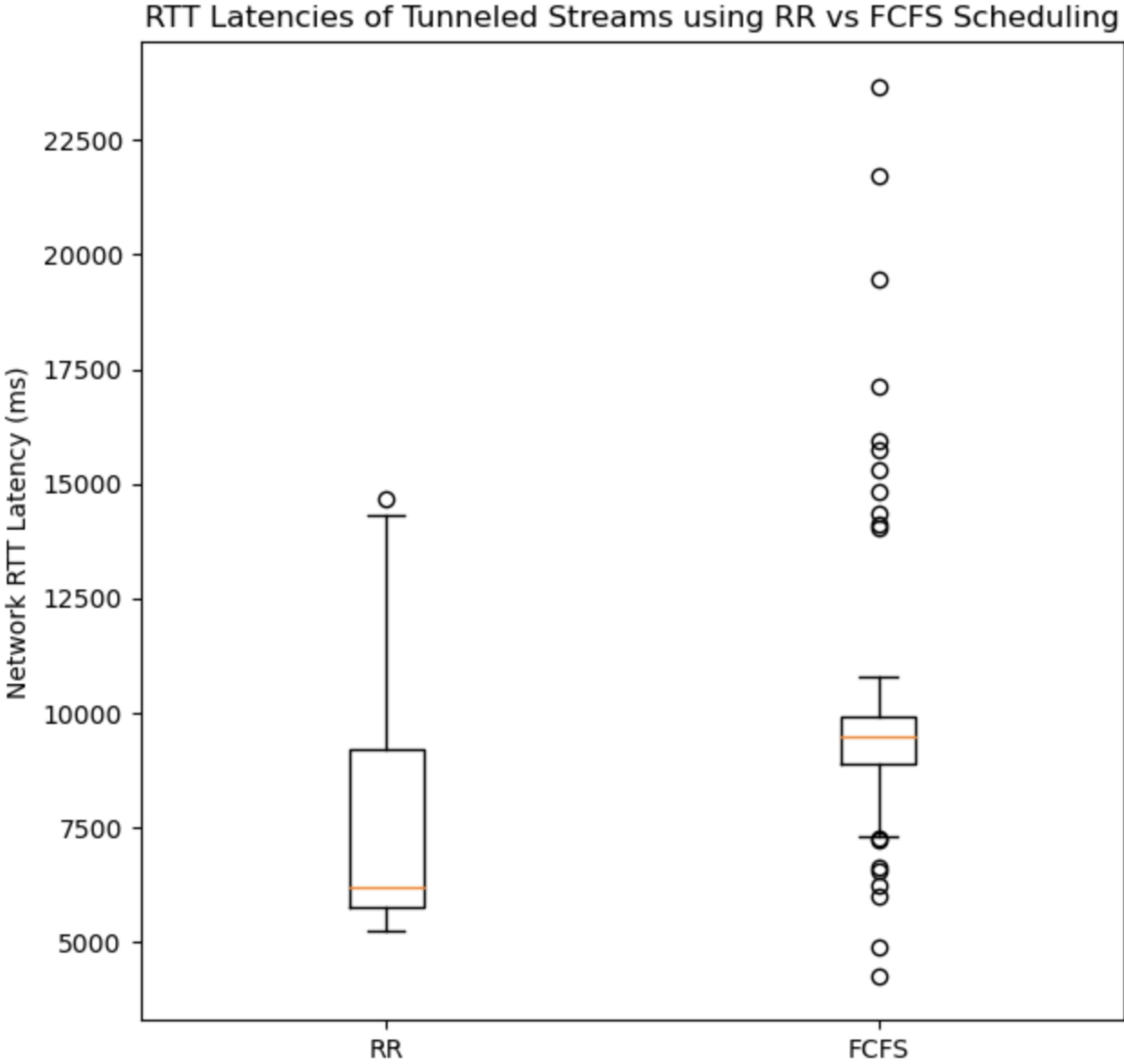

Figure 2.7: RTT Latencies of Tunneled Streams using RR vs FCFS Scheduling

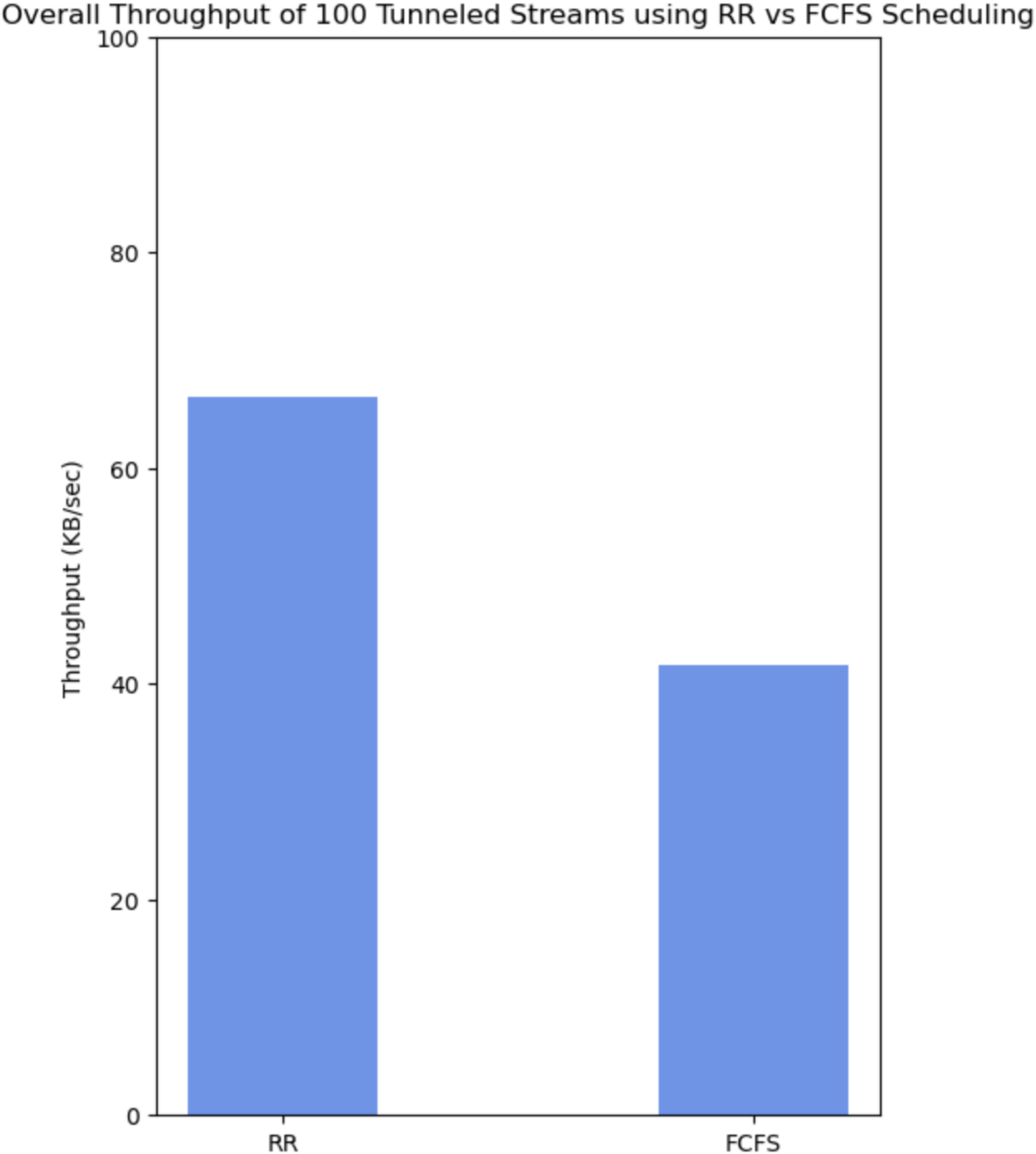

Figure 2.8: Overal Throughput of 100 Tunneled Streams using RR vs FCFS Scheduling

Priority Scheduling for Nested Streams

The second benchmark evaluates our stream scheduling implementation in capturing nested stream priority semantics. Since the inner and outer connections in MASQUE can be handled like any other QUIC connection, the stream scheduling mechanisms remain unchanged from the previous chapter. To indicate nested priorities, we designate the scheduler type during connection establishment for both outer and inner connections. Additionally, we assign the stream's priority at stream-creation time for both outer and inner streams. This method provides a hierarchical scheduling approach, as opposed to scheduling directly among all inner streams. Initially, the outer stream (inner connection) undergoes scheduling, followed by streams within that connection. This allows us to respect priority semantics at the connection level first, followed by the stream level. Governing the time slice allocation to each connection is beyond the scope of our work. Since both levels of streams are only visible to the client, it is responsible for managing nested scheduling semantics. The proxy and target servers are only responsible for scheduling one layer of streams each.

We open 10 end-to-end connections to the target server and 30 inner streams inside each of the 10 connections (for a total of 300 inner streams). This creates a nested stream structure wherein each of the 300 inner streams belongs to the flows of the 10 outer streams. We assign each of the 10 groups a relative priority value from 1 to 10 (with 10 being the highest, and 1 being the lowest). We schedule the inner streams using RR and compare between RR, FCFS, and ABS (absolute priority) for the outer streams. We assess the effectiveness of each scheduler in capturing the priority semantics among the outer streams, thereby validating our stream scheduling implementation on nested stream structures. Once again, we dial in 5 Mbps of bandwidth, 0.5% packet loss, and 50 ms of added packet latency with the tc command. However, we only send 1 KB of data per inner stream.

In Figure 2.9, we plot the mean inner stream round trip latency for each priority group for all three schedulers. As expected, the absolute priority scheduler effectively captures the priority semantics of the outer streams. The highest priority groups achieve the lowest round trip latencies of approximately 4000 ms. Mean round-trip latencies exhibit an almost monotonic decrease as priority levels increase. However, exceptions are observed with groups 9 and 10 - it's worth noting that round-trip latency is influenced by scheduling and retransmission events at the target server, which is oblivious to the proxy semantics of the outer streams. The network latencies for the RR and FCFS schedulers are quite comparable, with FCFS often performing better for several groups. Due to the reduction of packet loss rate and payload size transmitted through the network, there is far less congestion at the proxy compared to the previous benchmark. Consequently, the FCFS scheduler tends to outperform Round Robin, particularly in terms of mean latency.

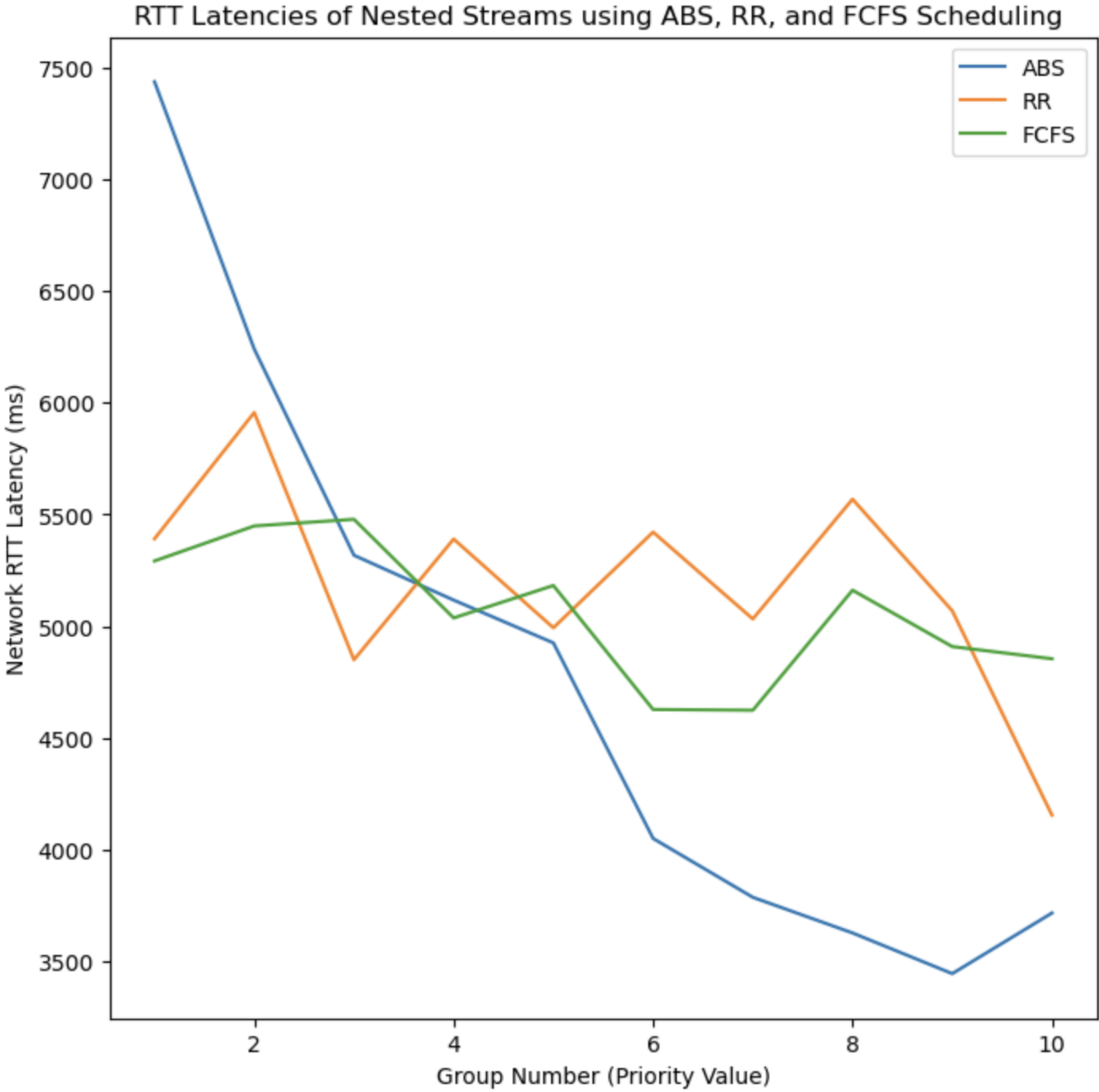

Figure 2.9: RTT Latencies of Nested Streams using ABS, RR, and FCFS Scheduling

PIFO Scalability Benchmarks

In the third benchmark, we compare our PIFO implementation against the base Absolute Priority scheduler, evaluating their performance on round-trip latency and scalability of stream management overhead. In the base version, we utilize a sorted list to order the streams by priority. To insert a new stream, we would have to compare priorities with a scan, a costly operation.

To accurately assess the impact of the PIFO scheduler, we restrict our architecture to a simple client-server model. We open 1000 streams to the server, assigning each a relative priority value from 1 to 10 (10 being the highest, 1 being the lowest). Each stream sends 10 KB of data to the server, which echoes the data back to the client. In addition to tracking round trip latency, we also maintain a tally of the total amount of time spent in stream management tasks, which encompasses the overhead of adding a stream to the queue, extracting it to packetize data, and subsequently reinserting it back into the queue. We dial in 5 Mbps of bandwidth, 0.5% packet loss, and 50 ms of added packet latency with the `tc` command.

The PIFO scheduler generally outperforms the absolute priority scheduler for most priority groups, based on Figure 2.10. This is especially apparent at the tails, with the max latency of the PIFO scheduler consistently lower than the max latency of the base scheduler for all groups. In most groups, the highest latency of the PIFO scheduler is lower than the 75th percentile latency of the base scheduler, demonstrating increased efficacy for high-latency bottleneck streams. The streams with the highest latency typically coincide with times of peak congestion, when the scheduling queue may contain hundreds of streams. Leveraging the superior asymptotic and runtime performance of the priority queue in PIFO, we efficiently schedule these high-latency streams, significantly reducing the overhead associated with stream management. Figure 2.11 compares the stream management overhead in terms of total time spent on stream insertion, extraction, and re-insertion. In the PIFO scheduler, stream management tasks collectively consume 9.87 ms, compared to 35.2 ms in the base priority scheduler for identical operations. We anticipate that the difference in overhead between the PIFO and base schedulers will amplify significantly as the number of streams further scales.

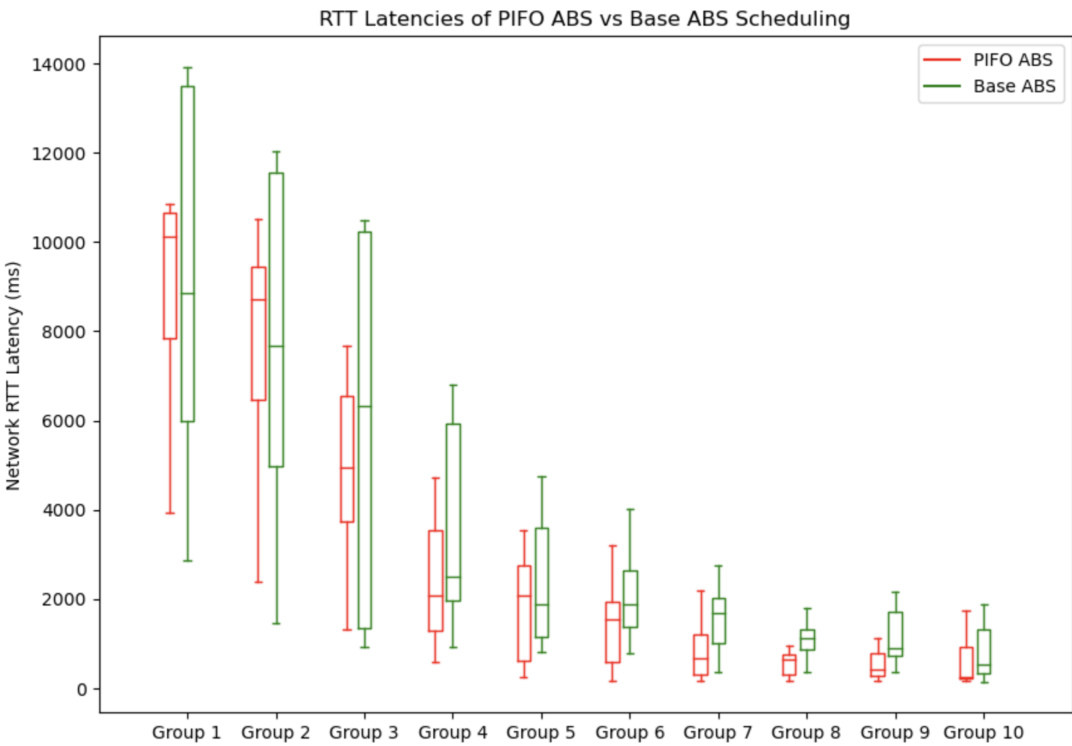

Figure 2.10: RTT Latencies of PIFO ABS vs. Base ABS Scheduling

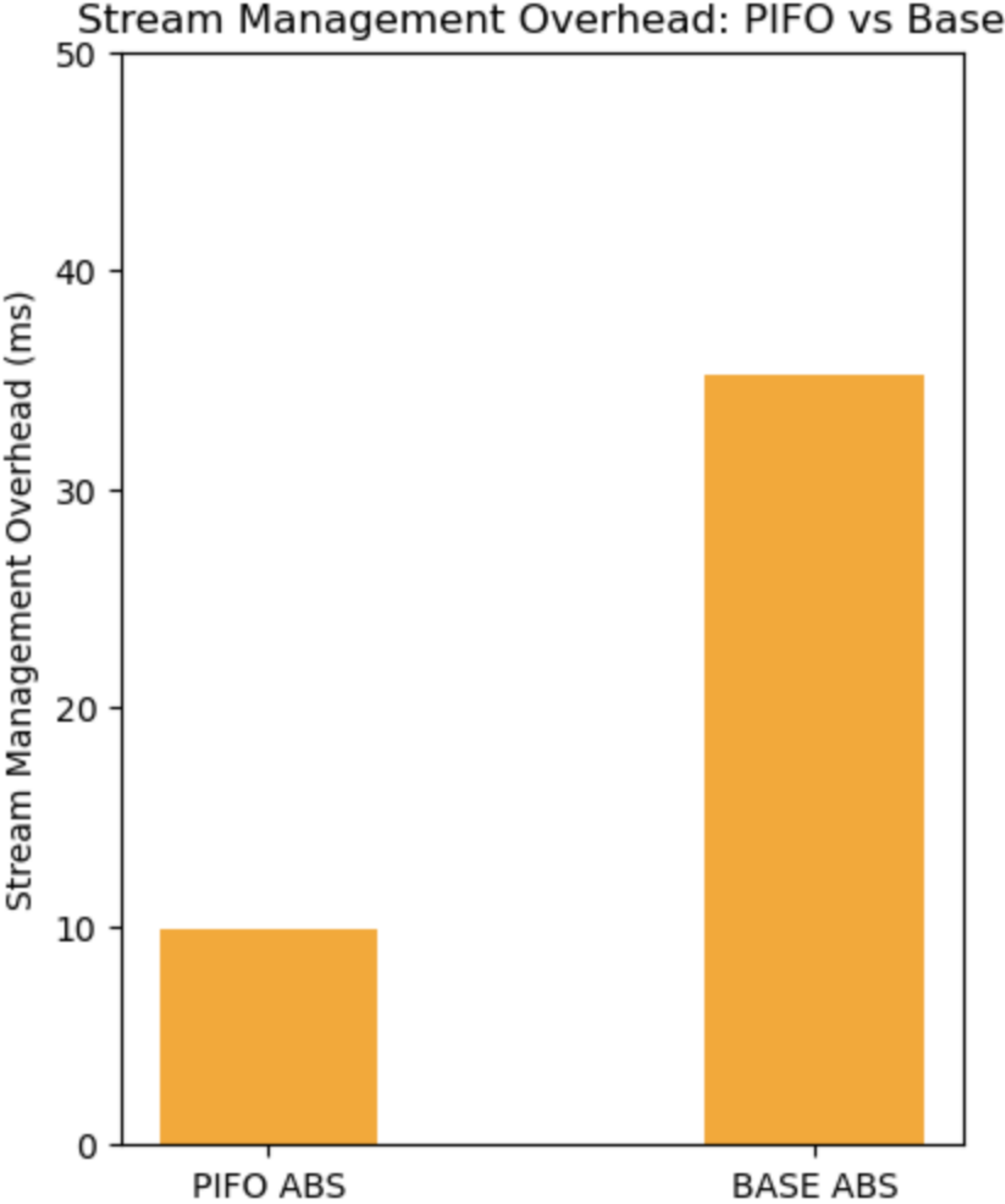

Figure 2.11: Stream Management Overhead: PIFO ABS vs. Base ABS

2.5 Future Work

There are many future directions to explore for stream scheduling in MASQUE. We outlined one such direction earlier, describing how QUIC Connection migration enables seamless and efficient transitions between proxies, enhancing overall end-to-end connection performance. Another avenue for future work involves implementing an RTT-based scheduling approach at the proxy level. In scenarios where target servers are situated at different distances from the proxy, closer servers may receive a disproportionate share of bandwidth allocation, especially when using RTT-influenced congestion control algorithms. Integrating RTT considerations into stream scheduling could improve fairness in this situation. Furthermore, we are currently exploring the feasibility of a hardware-based MASQUE implementation, where tunneling functionality is provided with programmable switches. As previously mentioned, tunneling operations involve extensive packet and header processing, tasks that can be efficiently handled at line rate using P4 [3] and programmable switches. This approach promises substantial improvements in packet latency and throughput.

2.6 Conclusion

In this paper, we have presented an architecture for MASQUE proxying and its implementation in `quic-go`. We have conducted a review of the current MASQUE literature, identifying and summarizing key unresolved issues, particularly in the context of stream scheduling. We benchmark the performance of various stream scheduling algorithms, demonstrating notable performance improvements over naive scheduling approaches. We also verified that our scheduling implementations effectively capture the hierarchical priority structure inherent in MASQUE's nested streams. Moreover, we implement the PIFO data structure, enhancing stream scheduling performance at scale. We show that the PIFO-based priority scheduler improves the scalability of stream management overhead, especially for bottleneck tail-latency streams.

Bibliography

[1] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. "SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 59–76. ISBN: 978-1-939133-13-7. URL: `https://www.usenix.org/conference/nsdi20/presentation/alcoz`.

[2] Albert Gran Alcoz et al. *Everything Matters in Programmable Packet Scheduling*. 2023. arXiv: `2308.00797 [cs.NI]`.

[3] Pat Bosshart et al. "P4: programming protocol-independent packet processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: `10.1145/2656877.2656890`. URL: `https://doi.org/10.1145/2656877.2656890`.

[4] L Clemente and M Seemann. *quic-go: A QUIC implementation in pure Go*. Accessed: 12/14/2023. 2023. URL: `https://github.com/quic-go/quic-go`.

[5] Apple Inc. *iCloud Private Relay Overview: Learn how Private Relay protects users' privacy on the internet.* 2021. URL: `https://www.apple.com/privacy/docs/iCloud_Private_Relay_Overview_Dec2021.PDF` (visited on 04/26/2024).

[6] Mirja Kühlewind et al. "Evaluation of QUIC-based MASQUE proxying". In: *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*. EPIQ '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 29–34. ISBN: 9781450391351. DOI: `10.1145/3488660.3493806`. URL: `https://doi.org/10.1145/3488660.3493806`.

[7] J Lainé. *aioquic*. Accessed: 04/27/2024. 2024. URL: `https://github.com/aiortc/aioquic`.

[8] Lucas Pardue and Christopher Wood. *Unlocking QUIC's proxying potential with MASQUE*. 2023. URL: `https://blog.cloudflare.com/unlocking-quic-proxying-potential` (visited on 04/26/2024).

[9] Tommy Pauly. "QUIC usage at Apple". 3rd Workshop on the Evolution, Performance and Interoperability of QUIC (EPIQ 2021). Dec. 7, 2021. URL: `https://epiq21.github.io/`.

[10] Tommy Pauly, Eric Kinnear, and David Schinazi. *An Unreliable Datagram Extension to QUIC*. RFC 9221. Mar. 2022. DOI: `10.17487/RFC9221`.

[11] Diego Perino, Matteo Varvello, and Claudio Soriente. "ProxyTorrent: Untangling the Free HTTP(S) Proxy Ecosystem". In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 197–206. ISBN: 9781450356398. DOI: 10.1145/3178876.3186086. URL: https://doi.org/10.1145/3178876.3186086.

[12] Miloslav Pojman. "How I wrote a Python client for HTTP/3 proxies". EuroPython 2022. July 14, 2022. URL: https://pojman.cz/2022/masque/.

[13] David Schinazi. *Proxying UDP in HTTP*. RFC 9298. Aug. 2022. DOI: 10.17487/RFC9298. URL: https://www.rfc-editor.org/info/rfc9298.

[14] David Schinazi and Lucas Pardue. *HTTP Datagrams and the Capsule Protocol*. RFC 9297. Aug. 2022. DOI: 10.17487/RFC9297. URL: https://www.rfc-editor.org/info/rfc9297.

[15] Justine Sherry et al. "Making middleboxes someone else's problem: network processing as a cloud service". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–24. ISBN: 9781450314190. DOI: 10.1145/2342356.2342359. URL: https://doi.org/10.1145/2342356.2342359.

[16] Anirudh Sivaraman et al. "Programmable Packet Scheduling at Line Rate". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 44–57. ISBN: 9781450341936. DOI: 10.1145/2934872.2934899. URL: https://doi.org/10.1145/2934872.2934899.

[17] Jeongseok Son. "Proxying HTTP/3 (QUIC) with CONNECT-UDP in Envoy". KubeCon + CloudNativeCon North America 2023. Nov. 6, 2023. URL: https://colocatedeventsna2023.sched.com/event/1Rj5y/proxying-http3-quic-using-connect-udp-with-envoy-jeongseok-son-google.

[18] Nikita Tyunyayev. "Improving the performance of picoquic by bypassing the Linux Kernel with DPDK". PhD thesis. UCL - Ecole polytechnique de Louvain, 2022. URL: http://hdl.handle.net/2078.1/thesis:37954.

[19] Zhaoguang Wang et al. "An untold story of middleboxes in cellular networks". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 374–385. ISSN: 0146-4833. DOI: 10.1145/2043164.2018479. URL: https://doi.org/10.1145/2043164.2018479.

[20] Nicholas Weaver et al. "Here Be Web Proxies". In: *Passive and Active Measurement*. Ed. by Michalis Faloutsos and Aleksandar Kuzmanovic. Cham: Springer International Publishing, 2014, pp. 183–192. ISBN: 978-3-319-04918-2.