Towards Machine Learning for Network Optimization: Buffer Sizing via Reinforcement Learning

Artemis Veizi
Advised by Professor Maria Apostolaki

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical and Computer Engineering
Princeton University

April 2023

I hereby declare that this Independent Work represents
my own work in accordance with University regulations.

Artemis Veizi

Towards Machine Learning for Network Optimization: Buffer Sizing via Reinforcement Learning

Artemis Veizi

Abstract

Buffer management helps to optimize network performance by controlling the flow of data and preventing congestion, which can lead to delays and packet loss. Buffer size is a critical aspect of the buffer management problem, as it determines how much data can be stored and processed in a network device. Advancements in technology have made it possible to obtain larger buffer sizes for network devices at lower costs, but this does not necessarily mean that increasing buffer size is always the best solution for network optimization; increasing the buffer size beyond a certain limit can lead to "bufferbloat", where excessive buffering causes increased latency, jitter and packet loss, and ultimately degrades network performance. Previous works have aimed to characterize the ideal buffer size for a given network under various network & traffic conditions, but lack extensibility to all topologies and network profiles. Furthermore, existing buffer management schemes set buffer thresholds according to fixed heuristic algorithms; static and heuristic buffer management algorithms suffer from weak adaptability to rapidly changing network traffic profiles, and often exhibit poor burst tolerance and significant throughput degradation in times of network congestion.

We present a reinforcement learning algorithm to select optimal buffer thresholds, with the aim of developing a buffer management strategy which more dynamically responds to changes in network traffic. We use NS–3 (Network Simulator 3) to simulate different network configurations with varying traffic loads, TCP protocols, and per-priority queue weights on a fixed topology, and compare our reinforcement learning algorithm (RLBM) to a statically-configured buffer management scheme (SB). Within a limited topological scope, simulation results indicate that RLBM produces the same or better throughput as SB in simulations with larger physical buffers. The RLBM scheme also showed significant improvement in the worst observed FCT slowdown and end-to-end delay for small buffer sizes. Our findings indicate that reinforcement learning algorithms could improve network performance over traditional buffer management schemes, and warrant further exploration of reinforcement learning solutions to the buffer management problem.

Acknowledgements

Thank you to Professor Maria Apostolaki for your guidance this year—it has been invaluable and deeply instructive. You have made this thesis one of the most rewarding learning opportunities in my time at Princeton. Thank you also to the Princeton University Center for Statistics and Machine Learning and Microsoft Corporation for generously gifting Azure Cloud Computing Credits, which were critical in the completion of this work. Lastly, thank you to Haiyue Ma and Hengrui Zhang for their willingness to share their work and support my thesis work.

Thank you to everyone in my 4 (...or 5) years that has helped me through the long problem sets, hard projects, hectic Dean's Dates, and bad days. More importantly, thank you to everyone who made the good days better—I hope I have reciprocated.

Thank you to Princeton Varsity Women's Lightweight Rowing, Coach Paul, and the entire Boathouse for the experience of a lifetime. It has been an honor to work towards a common goal with you all. P130 has defined my Princeton experience—it is something only we will know, and I will remember it forever.

Thank you to Sarah and Daisy for being the best friends and roommates a girl could ask for. When I think of Princeton, I will always think of Henry 226 first and most fondly—I can't wait for the lifetime of friendship ahead of us.

Thank you most of all to Mariana and Astrit Veizi, my endlessly supportive, loving, and kind parents. I love you so much. You have always believed in my potential, even when I could not. This thesis is the culmination of an 18–year journey in education that was only made possible by your hard work in fostering my love for learning & my belief in myself—to make you proud is my most worthwhile aspiration.

I tell you, my friend, all happiness depends on courage and work. I have had many periods of wretchedness, but with energy and above all with illusions, I pulled through them all. —Honoré de Balzac

Contents

1 Background

In order to optimize network performance, reduce congestion, and ensure reliable packet delivery, the Internet employs a range of protocols and algorithms that operate at different layers of the network stack. Some of the critical components of the Internet's infrastructure are Transmission Control Protocol (TCP), Active Queue Management (AQM), and Buffer Management (BM), which will be a central focus of this thesis.

1.1 TCP Congestion Control

Transmission Control Protocol (TCP) is a transport-layer protocol for computer networks which connects applications running on end hosts and enables packet transmission between them. TCP handles various aspects of the Internet communication model, providing reliable, ordered, and error-checked delivery of data between applications running on end hosts connected by an IP network; one of the important functions of TCP is flow and congestion control. The objective of TCP congestion control algorithms is to achieve high network throughput while avoiding network congestion and ensuring fairness among competing flows.

Generally, TCP congestion control algorithms (CCAs) take network state information as input (implicitly communicated from destination ACKs) to manipulate output parameters (namely, window size) in an effort to reduce congestion on network end hosts. These algorithms operate end-to-end: the end hosts are the ones responsible for implementing a particular congestion control algorithm. There are multiple algorithms which are commonly used in the Internet, which optimize on network efficiency and fairness [8].

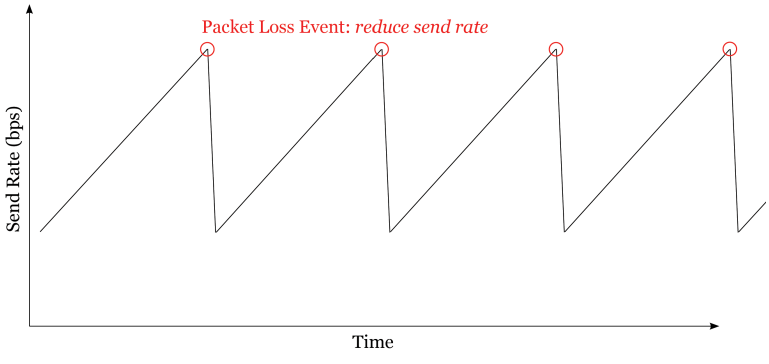

Figure 1: TCP congestion control algorithms continually adjust the packet send rate of the node to accommodate congestion; when a packet loss event is detected in the network, TCP will dramatically reduce the send rate to avoid more packet drops. Conversely, when packets are successfully delivered, TCP moderately increases the send rate to make best use of the network's bandwidth. This gives rise to a "sawtooth" pattern of transmission, which is characteristic of TCP congestion control.

Though implementation details vary across different congestion control algorithms, they all fundamentally regulate data transmission according to feedback about network congestion, which involves information like packet loss or delay; this is illustrated in Figure 1. In general, when a network event signals congestion (via packet loss or high delay), a TCP CCA will reduce its sending rate to avoid exacerbating the problem. Similarly, when no adverse network events are detected (no packet drops, low delay), TCP will increase the node's packet transmission rate to make more efficient use of the available bandwidth. More specifically, most TCP congestion control uses as an "additive increase, multiplicative decrease" algorithm:

- A *loss-based* congestion control algorithm will additively increase the packet transmission rate until packet loss is detected, at which point it will multiplicatively decrease the packet transmission rate to a "safe zone" in which packet losses are no longer experienced.
- A delay-based algorithm will behave similarly, but respond to increasing delay rather than packet loss; delay-based algorithms behave "predictively" by *avoiding* packet loss and network congestion, but may suffer from lower throughput as a result.

The choice of TCP protocol on end hosts therefore has significant impact on network traffic profiles.

1.2 Active Queue Management (AQM)

In networked nodes, packets that are received and transmitted are stored in queues until the end host is able to process them. Active Queue Management (AQM) is a set of techniques used to manage the queues in network nodes, such as routers and switches, to improve the performance and stability of the network. These algorithms are primarily concerned with the policy applied to enqueueing/dropping packets from queues on a given node; for example, the simplest approach to this is the Drop-Tail Queue, which simply drops packets when the queue is full, and accepts them at the end of the queue otherwise. AQM strategies can have an impact on throughput and flow completion, by purposely dropping certain packets even before the queue is full and allocating more or less queue space according to the priority of a given flow.

Most AQM algorithms work by monitoring the size of the queue and adjusting the drop probability of incoming packets based on the observed queue size. When the network is congested and the queue becomes "full" (or exceeds a certain threshold), an AQM algorithm will selectively drop packets from incoming TCP flows to signal congestion to the sending hosts. This packet drop event serves as feedback to the TCP congestion control algorithms on the sending hosts, which will then reduce their packet transmission rate and avoid further congestion.

The goal of AQM algorithms is to improve network performance and stability by preventing overflow in network queues, reducing packet loss, and signaling congestion to senders. AQM algorithms work in tandem with TCP CCAs to help achieve high network throughput, low latency, and fairness among competing flows. Because independent network nodes may implement different AQM strategies, there is another layer of complexity in analyzing the behavior of a network profile: in addition to the topology and the traffic pattern of a given network, the TCP protocol and AQM strategy implemented at each node will have significant impact on the rate and shape of packet transmission, as well as the completion time and distribution of all incoming flows.

1.3 Buffer Management

In internet networks, routers act as packet switches which transmit and receive packets to and from multiple end hosts. As packets are received, they are stored in the on-chip memory of the switch, which is shared between all end host queues (in the "shared buffer"). Buffer management involves allocating buffer resources to different traffic flows, prioritizing packets, and determining when to drop or discard packets in case of buffer overflow. The buffer size, per-queue buffer allocation, packet drop policy, and other parameters are manipulated by buffer management algorithms; in general, the goal of a buffer management algorithm is to maximize throughput (achieve high buffer utilization, avoid bufferbloat) and minimize packet drops (achieve adequately large buffer to handle incoming traffic, prevent overflow).

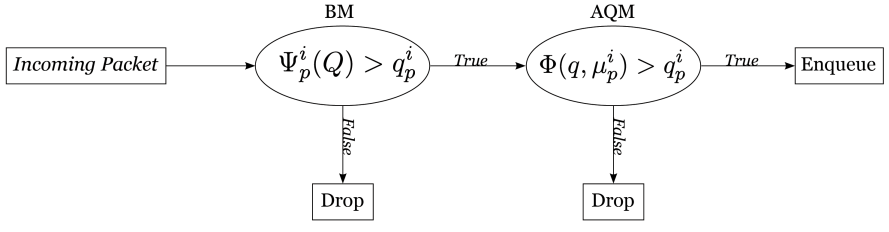

Figure 2: Buffer Management (BM) and Active Queue Management (AQM) work independently to make decisions about whether a switch buffer will accept (enqueue) an incoming packet, or drop the packet to provide feedback to the network about congestion. This illustrates a possible interaction between some BM and AQM algorithm; here, Ψ is the function the BM scheme uses to calculate the per-queue thresholds in the buffer, and Φ is the function the AQM scheme uses to calculate thresholds at its queue [2].

Buffer management plays a critical role in Internet networks because it helps to ensure that packets are delivered efficiently, with minimal delay and loss, and in a fair and efficient manner. The buffer on a given switch is shared between all senders. In

a shared buffer scheme, all sent packets are stored in the buffer, with occupancy limits assigned according to one of two possible schemes: (1) each queue is allocated a maximum possible amount of buffering (static threshold), or (2) packets are stored in the buffer until it is filled, at which time packets are pushed out (evicted) according to some selection heuristic (for example, evicting the packet at the head of the longest queue). Most buffer management algorithms work by manipulating the allowed buffer threshold based on input about (1) total remaining buffer space and (2) total remaining queue space (per-queue). This is the key interaction point between buffer management algorithms and active queue management algorithms, further illustrated in Figure 2 above.

A key aspect of the buffer management problem is buffer sizing. While a smaller buffer is desirable for higher throughput, the buffer should also be large enough to handle a bursty moment. Appenzeller et. al. conducted simulations to establish the appropriate size of a buffer, finding that "a link with n flows requires no more than

$$B = (\overline{RTT}) \times C)/\sqrt{n}$$

for long-lived or short-lived TCP flows" to achieve 98% utilization [3]. This work is discussed further in Section 2.2, Related Work: Sizing the Buffer). A significant emphasis of the work on buffer management so far has been to determine the optimal size of the buffer to prevent congestion without wasting resources; as a network parameter, buffer size offers a continuous, fine-grained axis of control which makes it an attractive target for hands-off, algorithmic manipulation. For these reasons, buffer size is a reasonable parameter to target for control via a reinforcement learning algorithm.

1.4 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning approach in which an agent uses trial-and-error outcomes to learn how to make optimal decisions in a given system. In RL, an agent interacts with an environment (selecting an action from some constrained possible action set), receives feedback in the form of rewards or penalties (which are determined by the *state* of the system), and learns to maximize the cumulative reward over time by selecting actions that lead to higher reward outcomes.

At a high level, RL models operate by mapping the agent's observations of the environment to actions. The RL Agent's "policy" is a function that takes the state of the environment as input and produces some probability distribution over the possible action set. The agent then selects an action from this distribution based on some algorithmic exploration strategy, waits for some time delta for this action to take effect in the environment, and then receives the resulting reward signal from the environment. This cycle repeats iteratively, with more and more of the environment's state space being explored as the agent develops and reinforces their policy—this cycle is illustrated in Figure 3.

The objective of reinforcement learning is to maximize the cumulative reward over time by iteratively updating its input-output mapping based on the observed rewards and states. This approach is characterized by its real-time, adaptive nature: reinforcement learning models have the potential to learn complex patterns in network data and adapt to changing network conditions, making RL a promising approach in improving network performance.

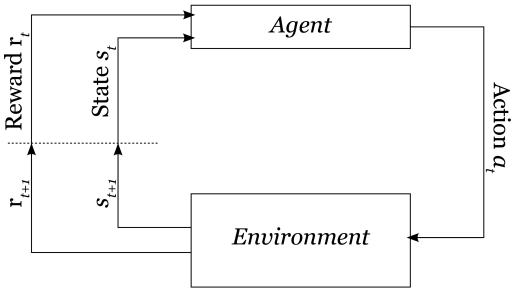

Figure 3: This workflow illustrates, at a very high level, the state-reward-action path of a reinforcement learning algorithm.

RL: Algorithmic Implementations At a high level, reinforcement learning algorithms can utilize a value-based method[1] or a policy-based method[2]. There are several possible algorithmic implementations of a reinforcement learning agent (we refer to the algorithm which makes decisions in the environment as the "RL Agent"). The following is a high-level overview of two reinforcement learning algorithms which have been applied in networking research for performance optimization (these approaches are discussed further in Section 2.4: RL-Based Approaches to Buffer Management):

- **Actor–Critic** Actor–critic algorithms combine both value-based and policy-based methods, by employing an "actor" which learns a policy, and a "critic" which evaluates the outcomes of the actor's actions.

  In this framework, the actor component learns the optimal policy by directly mapping states to actions, while the critic component learns to evaluate the goodness of each state by estimating its corresponding value. The critic is responsible for estimating the value function and providing feedback to the actor in the form of a state-value function; the actor then updates its policy based on this feedback to maximize its expected reward.

[1] A value-based reinforcement learning algorithm learns to estimate the optimal value of each state or state-action pair, and uses this information to make decisions.

[2] A policy-based reinforcement learning algorithm directly learns an optimal policy for making decisions, without explicitly estimating the value function; as a result, policy-based methods may be less computationally intensive than value-based methods, for which the value function might be expensive to compute for large state or action spaces.

Actor–critic algorithms employ two different neural networks, one for the actor and one for the critic. The actor network maps states to actions, while the critic network estimates the expected value of each state. The algorithm then computes the difference between the critic's value estimate and the actual reward of the selected action—this is referred to as the temporal difference error. This error calculation is then used to update the critic's value function, which in turn is used to update the actor's policy [5]. This dual-network system offers a balance between exploration and exploitation of the environment, making actor-critic algorithms a powerful tool for decision making in complex and dynamic environments.

- **Proximal Policy Optimization (PPO)** PPO is a policy gradient reinforcement learning method, and is specifically designed to optimize policies in continuous control and high-dimensional action spaces; a particular advantage of PPO is that it uses a clip function to ensure that updates to the policy are within a specified, limited range, preventing too-large policy updates that can destabilize the learning process and lead to poor performance; the clipping function effectively limits the distance between the old and new policies, ensuring that updates to the policy are within a reasonable range [7]. The PPO algorithm iteratively collects a batch of trajectories by following the current policy, calculates the advantages of each state-action pair, and then updates the policy using the clipped surrogate objective. The advantages are estimated by comparing the expected value of each state to the actual reward obtained, similar to actor-critic algorithms; this feature of the algorithm helps to improve sample efficiency, as the agent can learn to estimate the value of states without having to explore them explicitly.

  Ultimately, PPO is computationally efficient and improves upon policy gradient methods which can suffer from instability, making it an attractive choice of algorithm for our RL–based buffer management implementation.

This work focuses specifically on the potential of RL models to improve buffer management by learning a policy which manipulates buffer size to optimize a reward based on maximized network performance metrics (e.g. throughput, delay) and minimized packet loss. Such an RL agent could learn from states of the network environment to associate performance outcomes with buffer sizing decisions, and in turn, make better buffer sizing decisions in the future. The merit of this approach is its overall adaptability: the agent's learning pattern can take into account various factors that affect buffer sizing, such as network traffic patterns, application requirements, and network topology, without prior knowledge of these conditions (or how they might affect the "optimal" buffer size). Moreover, RL–based buffer sizing could potentially optimize network performance beyond what is achievable with a fixed heuristic approach to buffer sizing, particularly in complex or uncertain network environments.

Buffer Sizing: Algorithmic Considerations The action space (or "variable space") of a problem is a crucial consideration in selecting a reinforcement learning algorithm; firstly, the choice of algorithm depends on whether the chosen variable is continuous or discrete. Moreover, the complexity of the environment, the size of the state and action spaces, and the computational resources available to the agent all provide constraints on the choice of suitable RL algorithm.

PPO is an ideal choice for the buffer sizing problem because it is well-suited to continuous control problems, and has been shown to provide stable and effective results in complex environments like congested networks. Additionally, PPO's on-policy nature allows it to continuously learn from changes in the environment and improve its performance over time—this is vital for an algorithm which would theoretically be continuously applied at critical nodes in a network, where traffic patterns and network needs are continuously changing.

However, using RL for buffer sizing also presents some challenges; in general, RL–based solutions may be computationally expensive, particularly for large-scale networks, which can limit their practical applicability—even a computationally "lighter" method, like PPO, is still more intensive than a simple heuristic-based buffer management algorithm. Further research is needed to develop efficient and scalable RL–based solutions for buffer management in real-world networking environments.

2 Introduction

Given our discussion in Chapter 1: Background, we now turn to the current body of research in buffer management and reinforcement learning applications in networking, to ultimately motivate this thesis work.

2.1 Motivation and Theory

The problem of buffer management is of particular interest in internet networking, as the buffer can play a significant role in network performance outcomes. Traditional buffer sizing algorithms often rely on static "rules" or heuristics, which do not take into account the dynamic and complex nature of network traffic, and offer limited adaptability to changing network conditions or traffic patterns—this limitation can lead to performance degradation and network congestion under heavy traffic loads or moments of burstiness.

Reinforcement learning offers a promising approach to buffer management by allowing network operators to dynamically adjust buffer sizes based on real-time feedback from the network. An on-the-loop RL agent could learn a policy based on buffer sizing decisions and their corresponding network performance outcomes to make better buffer sizing decisions in the future, with the ultimate goal of optimizing network performance by minimizing packet loss and reducing congestion. This would also potentially enable better burst absorption, as the agent could adapt its strategy to accommodate a sudden burst on a short time scale.

The primary motivation for this approach is the complex nature of networking: the state space of possible network configurations and topologies is virtually infinite, and network traffic can be highly dynamic and unpredictable (this is discussed in detail in Chapter 3: Proposed Work). A reinforcement learning agent can learn complex policies which implicitly take into account various factors that affect buffer sizing decisions, such as network topology, application requirements, and traffic patterns, without enumerating each factor and its explicit impact on ideal buffer size. RL–based solutions can potentially improve network performance beyond what is achievable with traditional buffer sizing algorithms, and they can adapt to changing network conditions and traffic patterns, making them well-suited for the dynamic and complex nature of internet networking. This research aims to investigate the use of a reinforcement learning model to manage the buffer in internet switches, with the goal of achieving optimal network performance while minimizing the risk of congestion. Our research problem is to design and evaluate a reinforcement learning model that can learn to size the buffer effectively on short time scales, given only the network performance metrics that would be available in a real-world network.

We will now turn to a discussion of related work in buffer management and sizing algorithms, as well as the application of reinforcement learning in buffer management.

2.2 Related Work: Sizing the Buffer

Sizing Router Buffers The scope of this thesis work is focused on buffer sizing; our approach to the buffer management problem primarily builds on research conducted by Appenzeller *et. al*, which conducted simulations and real-network experiments to find the minimum amount of buffer necessary to maintain throughput for a given number of flows on a single link [3].

Buffer management algorithms select buffer size with the goal of ensuring a congested link is busy 100% of the time; this problem can be restated as *ensuring that the buffer never goes empty*. Appenzeller *et. al.* claim that the buffer size rule-of-thumb $B = \overline{RTT} \times C$ is outdated and, more specifically, incorrect for backbone routers due to the "large number of flows multiplexed together on single backbone link"[3]. Though it is still true that most traffic uses TCP, the number of flows has significantly increased, motivating a need to establish a better (or more up-to-date) rule of thumb for buffer size. Given current network conditions, it is not understood how much buffering is actually needed, nor how it impacts network performance.

Overbuffering "complicates the design of high-speed routers, leading to higher power consumption, more board space, and lower density" [3]. Overbuffering also "increases end-to-end delay in the presence of congestion"—large delays can make CCAs unstable and applications unusable [3]. The goal of buffer sizing is to prevent overflow; however, TCP congestion control algorithms often behave in such a way as to fill the buffer and deliberately cause overflow/packet loss. So, an additional goal in selecting the buffer size is to prevent underflow in TCP flows—the key metric for underflow is throughput (utilizing less than the maximum bandwidth is "underflow"); the goal for Appenzeller *et. al.* was to maximize throughput at the bottleneck link.

Problem Statement: For *backbone routers*, what is optimal buffer size to maximize throughput and minimize end-to-end delay?

Solution: Appenzeller *et. al.* found that a link with n flows requires no more than

$$B = (\overline{RTT} \times C)/\sqrt{n}$$

for both long- and short-lived TCP flows.

A limitation of these findings was that the ideal buffer size B was established for a particular set of network conditions: senders were configured with the TCP Reno congestion control algorithm and a single drop-tail FIFO queue (though it is expected that the results would extend to the AQM Algorithm RED).

2.3 Related Work: Active Buffer Management

ABM: Active Buffer Management in Datacenters A recently published work by Addanki *et. al.* puts forth Active Buffer Management (ABM) [2] as a new strategy for managing the buffer, which improves upon existing buffer management algorithms. The

limitation of heuristic buffer management algorithms is that they react to congestion *after* it occurs by reducing the incoming data rate, which can result in low network utilization and increased delay. On the other hand, static buffer management algorithms allocate a fixed buffer size to each port, which can result in underutilization or overflows depending on the traffic pattern. Both reactive and static buffer management algorithms fail to dynamically adjust buffer thresholds based on network congestion and do not offer proactive buffer management—ABM is an alternative approach that aims to improve network performance by proactively managing buffer space to prevent buffer overflows.

ABM involves monitoring the buffer space in real-time and taking action to prevent buffer overflows before they occur. The ABM algorithm uses a feedback control loop to adjust the buffer space dynamically, based on the observed workload and buffer utilization; this is designed to balance the trade-off between buffer utilization and queuing delay, in order to optimize overall system performance.

Moreover, this approach aims to resolve the discordance between Active Queue Management (AQM) and Buffer Management (BM) which is used in typical hierarchically-organized networking devices—this disparity occurs because the BM scheme sets the maximum buffer size, and the AQM scheme independently sets the packet enqueue policy at each queue, resulting in interference across queues, increased queuing delay, and reduced burst tolerance [2]. To address this common flaw, the authors propose a strategy which incorporates both buffer occupancy and queue drain time in the calculation of buffer size, thereby integrating strategies from both BM and AQM schemes. ABM assigns a threshold $T_p^i(t)$ to the queue of priority p at port i as follows:

$$\boxed{T_p^i(t) = \alpha_p \cdot \frac{1}{n_p} \cdot (B - Q(t)) \cdot \frac{\mu_p^i}{b}}$$

where [2],

- α_p is the operator-assigned alpha (weight) for the priority class p.
- n_p denotes the number of congested queues of priority p.
- $B - Q(t)$ denotes the available remaining buffer space.
- μ_p^i denotes the normalized drain rate for the queue of priority p at port i, and
- b denotes the bandwidth per port.

The authors show that ABM offers isolation across all priorities, such that no single priority can starve out other (lower) priority queues. Furthermore, ABM bounds the queue drain time by taking into account the drain rate of each queue. Crucially, as a result of these properties, ABM offers predictable burst tolerance, which most buffer management schemes cannot guarantee. The results show that ABM outperforms existing techniques in terms of both throughput and latency.

Though this state-of-the-art scheme addresses many of the shortcomings of current buffer management schemes, it still accomplishes this in a heuristic manner, by implementing more network metric coefficients in a fixed algorithm to set the queue thresholds. This still presents a somewhat limited approach, as these coefficients are not always readily available in real-world Internet networks, and the heuristic-based approach is still less adaptable to network conditions than what might be possible with reinforcement learning.

2.4 Related Work: RL-Based Approaches to Queue & Buffer Management in Networked Environments

LFQ: Online Learning of Per-flow Queuing Policies using Deep Reinforcement Learning Queuing policies play a crucial role in network traffic management by prioritizing and scheduling packets for transmission, while balancing various competing factors such as network congestion, latency, and fairness. One approach is per-flow queuing (PFQ), in which traffic flows are separated and queued based on their priority levels; however, the allocation of priority levels to flows can be challenging, and traditional methods often require significant manual tuning and adjustment.

Deep reinforcement learning (DRL) has emerged as a promising technique for automated decision-making in complex and dynamic environments like networks; in brief, DRL algorithms learn to make decisions by interacting with the environment and receiving feedback in the form of rewards. This dynamic quality of DRL algorithms motivates their use in addressing current challenges with per-flow queueing.

In a recent work by Bachl *et. al.* [4], a novel DRL–based approach was proposed for online learning of PFQ policies. The authors present Learning Fair Qdisc (LFQ), an online actor–critic algorithm that learns to allocate priority levels to network flows based on their observed characteristics, such as the size and arrival rate, and evaluates actions on whether the choice of buffer size was better or worse than expected. The algorithm uses a neural network architecture to represent the PFQ policy, and authors conducted testing for both an online (updated in real-time based on the observed rewards and network state) and offline policy.

The authors evaluated the performance of LFQ in simulations; results showed that LFQ outperformed traditional PFQ methods (fq, FqCoDel) and achieves near-optimal performance in terms of throughput and delay. LFQ outperformed all the compared methods in terms of the average throughput, and consistently demonstrated lower average queue length, indicating a more robust response to changes in network topology and traffic patterns.

The results of this study show that LFQ can adapt to changing network conditions and learn effective PFQ policies on the fly, demonstrating the potential of DRL for automated network traffic management, and, more generally, the viability of RL–based approaches in network traffic management. These findings suggest that RL–based

approaches could significantly improve the performance and efficiency of network communication, warranting further research in this area.

Reinforcement Learning for Active Buffer Management A new study by Haiyue Ma and Hengrui Zhang proposes the use of reinforcement learning for active buffer management in networking systems; this work aims to address the inflexibility of traditional buffer management strategies, which are often unable to adapt quickly to the rapid changes in incoming traffic. The authors extend Active Buffer Management scheme discussed in Section 2.3, and propose the use of reinforcement learning to train the ABM equation parameters. They introduce a PPO-based RL agent which takes as input the current state of the buffer, outputs the queue length thresholds, and processes resulting flow efficiency metrics as rewards. The authors implemented their design in NS3 for testing and analysis, demonstrating performance results comparable or superior to ABM.

This thesis work will build primarily on the work accomplished by Ma and Zhang.

2.5 Project Outline

Having established the scope of this thesis work, we now turn to a discussion of our experimental approach. Further implementation details are presented in Chapter 3: Proposed Work. Generally speaking, the structure of the project was as follows:

1. Fix some topology T which generates congestion in networked nodes.
2. At the congested node, fix some buffer management strategy $B \in \{\text{RLBM, SB}\}$[3].
3. Run a fixed-length simulation on topology T running strategy B, and collect network performance data.
4. Compare performance outcomes of different buffer managers B on fixed configurations of topology T.

Using a fixed topology T for all choices of B allows us to draw valid comparisons between the various buffer management strategies, and in particular, will allow us to observe whether the reinforcement learning algorithm improves performance outcomes by selecting better buffer sizes. As discussed in Section 2.2, buffer size is a critical component of the buffer management problem: compellingly, it offers a continuous variable for the buffer management algorithm to manipulate, which is particularly relevant for the RL–based buffer management scheme[4]. Moreover, Active Buffer Management (discussed in Section 2.3) provides several standards of comparison in measuring the efficacy and performance outcomes of various buffer management schemes; this work

[3]Where RLBM denotes Reinforcement Learning Buffer Management, and SB denotes Static Buffer.

[4]Refer to Section 1.4 for a discussion of Proximal Policy Optimization and its application for continuous variables

will utilize many of the same mechanisms and measurements for comparison. Lastly, this work aims to contribute to the growing body of research exploring the integration of machine learning algorithms with networking. Previous work has shown the potential of machine learning algorithms to improve network performance; the goal of this project is to investigate the application of reinforcement learning to buffer management and contribute to this body of research.

3 Proposed Work

The problem of buffer management in internet networks presents a complex, high-dimensional parameter space which is virtually impossible to systematically characterize and challenging to understand. This thesis work explores a limited scope of the problem, reducing the parameter space of networked systems in order to provide a more comprehensive and detailed set of outcomes. We will now turn to a brief discussion of the extended networking problem space, in order to explain the particular scope of this project.

Networked Environments: Infinite State Space Internet networks are composed of a multitude of independent, interconnected components that interact with each other in complex ways; these components include routers, switches, servers, end-hosts, and other networking devices, each of which retains its own set of properties and configurables. In addition, network environments play host to different traffic profiles, with characteristics such as flow size, flow rate, and packet inter-arrival times; all of these variables together result in a high-dimensional characterization of any given network environment.

The following are some of the key variables that define network topologies and contribute to this high-dimensional problem space:

1. **Network Size** The number of devices and the scale of the network.
2. **Network Topology** The physical or logical arrangement of network devices; some examples include star topologies and leaf-spine topologies, though the number of possible topologies is theoretically infinite.
3. **Node Configuration** An individual network node may be configured in many different ways according to the needs of the user/network; these include its network address configuration, the link-layer and transport-layer protocols it employs, as well as any security protocols it requires, to name a few.
4. **Link Capacity** The bandwidth, or maximum data rate, that can be transmitted over a link, expressed in bits per second.
5. **Link Delay** The latency, or wall-clock time delay, experienced by data as it travels across the network from its source to its destination.
6. **Routing Algorithm** The algorithm used to determine the path that a packet takes through the network, from its source to the destination node.
7. **Traffic Type** The type of traffic being transmitted, such as voice, video, or data. Different networks experience different traffic profiles; for example, datacenters often deal with longer flows, whereas internet networks most often experience short flows and bursty, unpredictable traffic profiles.

8. **Traffic Size** The amount of traffic (in bytes, for example) flowing through the network at a given time.

9. **Algorithmic Decisions** Components in the network will utilize different algorithms to accomplish their work; for example, an end-host might employ a particular TCP Protocol (like TCP NewReno), a switch queue might employ a particular AQM (like CoDel), and more. Routing, congestion control, queue and buffer management, scheduling, and security all employ various algorithms which exhibit different behaviors in networked environments.

10. **Quality of Service (QoS) Requirements** The level of service required by applications, such as minimum delay, maximum jitter, or minimum throughput.

Moreover, network performance can be influenced by factors outside of the network itself, such as the physical distance between nodes, the quality of the network connections, and the characteristics of the devices and applications being used. This is not an exhaustive list of variables and factors that can impact network performance, but it should give an idea of how complex the problem space is: each of these variables can have a large range of possible values, resulting in an exponentially large number of possible network configurations.

Taken together, the number of variables involved in network performance is enormous, and it is virtually impossible to enumerate and explore them systematically to find optimal solutions for each configuration. As such, it is necessary to limit the scope of our research problem by reducing the network environment to a simple topology.

Limiting Scope: Reduced State Space To approach the problem of buffer management in internet networks, we will fix a particular topology, such that the variable space of the network environment is greatly reduced. This topology will enable us to simulate congestion, as we are interested in exploring buffer management schemes under high-traffic network conditions (where packet drop policies, congestion control algorithms, and other QoS control factors come into play). In our fixed topology, the variable space of our network environment will be:

1. **Node Configuration** Each node in the network will be configured with a particular TCP protocol (which will determine its congestion control algorithm). Furthermore, nodes on the congested link will be configured with a particular buffer management strategy, and their buffers/queues will be assigned a specific maximum physical size.

2. **Traffic Size** We will vary the load on the network to observe the response of the buffer management schemes under testing.

We now explore a parameter space which is reduced to

$$S(\text{BM, TCP, Network Load, Buffer\_MaxSize})$$

Given this reduced problem, we propose a two-part project which first attempts to characterize S under static buffer management schemes, and then explores the effect on S when a reinforcement learning algorithm is used to manage the buffer.

3.1 Characterizing the Buffer

The goal of this first part of our research is to understand how each network configuration S(BM, TCP, Network Load) responds to changes in Buffer_MaxSize, to better understand the optimal buffer sizes for different network configurations. This work specifically aims to build on the work by Appenzeller *et. al* to characterize the impact of various buffer sizes under different network conditions, with the ultimate goal of developing a recommendation of the optimal buffer size for multiple network configurations.

3.1.1 Static–Size Buffer

To accomplish this, we will use NS–3 (Network Simulator 3) to simulate different network configurations with varying traffic loads, TCP protocols, and per-priority queue weights on a fixed topology. In each simulation, we will vary the buffer size parameter to observe corresponding changes in network performance; our key performance metrics include the network throughput, delay, and packet loss rate for each simulation run. Using the results from these simulations, we can (within a limited scope) establish the "optimal" buffer size for each network configuration S, to serve as a baseline for comparison with the results obtained from the RL–based approach employed in later stages of the project.

The essential procedures for this part are as follows:

1. Build an NS–3 simulation S(Static Buffer, TCP, Load, Buffer_MaxSize) to run under varied TCP and static buffer configurations.
2. For each configuration, run S with varying buffer sizes.
3. Collect data on network performance metrics (e.g. throughput, RTT) for each simulation run.
4. Analyze the data to identify optimal buffer sizes for each network configuration.
5. Identify how different network configurations respond to changes in buffer size.

This portion of the project will also serve as a validation of the constructed simulation, before incorporating the reinforcement learning algorithm. The advantage of this approach is that it provides a systematic method to explore the impact of buffer size on network performance for various network configurations, ultimately enabling us to identify the network configurations that are more sensitive to buffer size changes and determine the optimal buffer size for each network configuration.

3.1.2 Limitations of this Approach

The main limitation of this first part of the project is that it is not extensible to all possible network configurations. While this approach provides valuable insights into how buffer size affects network performance for a variety of network configurations, it is limited to the specific set of configurations that are tested in the simulation under our fixed topology. The results obtained from these simulations cannot readily be extrapolated to other network topologies that are not explicitly tested. Therefore, while this approach will serve as a useful baseline for comparison with the RL–based approach, it is not a comprehensive solution for optimizing buffer size in all network configurations.

3.2 Buffer Management via Reinforcement Learning

The goal of this part is to create a reinforcement learning algorithm to manage the buffer, and compare its performance to the performance we observed in the first part of the project, where buffer size was static. This will allow us to evaluate the effectiveness of using an RL–based approach to buffer management, and determine if it is a viable solution for improving network performance. Ultimately, the aim is to build on our characterization of the buffer by identifying the optimal buffer size for different network configurations and traffic patterns dynamically, using RL.

To accomplish this, we will develop an RL agent which uses Proximal Policy Optimization to select the buffer size, and integrate it with our validated simulation environment S. As discussed in Section 1.4: Reinforcement Learning, PPO is a better choice for the buffer management problem because it is well-suited to continuous control problems (e.g. continuously varied buffer size) and has been shown to provide stable and effective results in complex environments. Additionally, PPO's on-policy nature allows it to continuously learn from changes in the environment and improve its performance over time, enabling us to create an online learning agent to manage the buffer in real time.

The essential procedures for this part are as follows:

1. Develop an RL–based buffer management algorithm using PPO in PyTorch.
2. Integrate the RL agent with our existing NS–3 simulations.
3. Run simulations S(RL Agent, TCP, Load, Buffer_MaxSize) with varying TCP and Buffer_MaxSize configurations.
4. Collect data on network performance metrics (e.g. throughput, RTT) for each simulation run.
5. Compare the performance of the RL buffer management scheme with the performance of the static buffer from part 1.

Overall, the proposed two-part plan aims to address the limitations of current approaches to buffer management in internet networks by leveraging the power of reinforcement learning. The first part of the project provides the foundation for this approach by validating the simulation and collecting data on how network configurations respond to changes in buffer size, while the second part of the project builds on this foundation by using the data collected in the first part as a benchmark for an RL algorithm that can dynamically adjust buffer sizes in real-time to optimize network performance.

4 Design and Implementation

A simple network topology was constructed with Network Simulator 3 to simulate buffer conditions in high-traffic Internet switches. At the intermediate node on the bottleneck link, the buffer was shared between all queues[5]. Two separate queues were implemented at this node, one each for Priority 0 and Priority 1 traffic, all of which shared the same port. By bandwidth-limiting the bottleneck link, we were able to simulate a high-traffic environment in times of congestion.

4.1 Network Simulator 3 (NS–3)

NS–3 is a discrete-event network simulator which offers the capability to simulate and study various communication networks, protocols, and applications [1]. The critical components of the NS–3 Object model are:

- `ns3::Node`, a highly generalized structure used to represent any network end point (host). By itself, a `Node` is a very basic computer with essentially no functionality.
- `ns3::NetDevice`, a class which can be layered onto a `Node` to enable its use as a network device; this essentially gives the "bare computer" a network card.
- `ns3::Application`, a class whose instances can be "installed" on a NetDevice to implement specific functionality, like traffic generation. Each `Application` can involve zero or more `Sockets`; each `Node` points to a list of `Applications`, and have multiple installed.
- `ns3::Channel`, a class which implements network communication channels. Each `Channel` offers the ability to attach multiple `NetDevices`, and provides a path for information to flow between them.

With these four basic objects, one can create and simulate various network topologies, configure network protocols & applications, and generate traffic patterns. The `Node` base object is illustrated in Figure 4.

NS–3 was an ideal choice to simulate the network environment for this thesis project, as it offers extensive monitoring capabilities: after constructing a topology, it is possible to collect and analyze simulation data, detect significant network events, and perform statistical analysis on simulation performance metrics. Moreover, NS–3's object model is designed to be modular and extensible, allowing the addition of custom components and features as needed. This extensibility enabled the development of fine-grain network monitoring, a highly-customized simulation topology, and ultimately, an online RL agent. We will now discuss the most relevant components of the NS–3 source library which were used in constructing the simulation environment for this thesis project.

[5]The buffer is used to store traffic that cannot be received at the same rate it is transmitted; in order to test various buffer management strategies on the buffer at this node, it was necessary to design a topology with such a bottleneck.

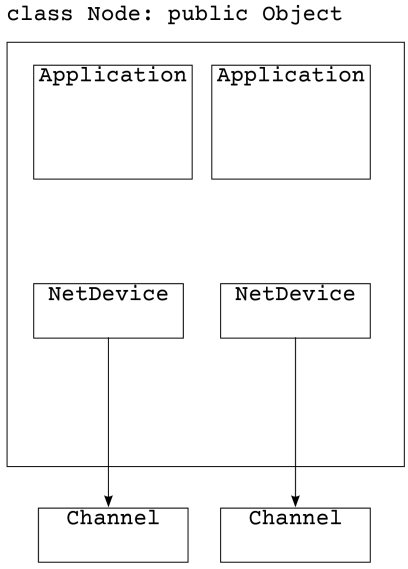

Figure 4: The `Node` class provides an implementation of a basic computer, on which various `NetDevices` and `Applications` can be installed, in addition to other specific capabilities (e.g. TCP protocol, Socket implementations, and more). Multiple `Nodes` can be connected to each other via shared channels, enabling communication between them. In general, communication between `Node` objects is accomplished via installed `Applications`, which generate traffic and send it to a specified end host.

4.1.1 NS–3: Queue Disciplines

The `QueueDisc` class is an abstraction included in the NS–3 library for implementing packet queuing and scheduling mechanisms in network nodes. The `QueueDisc` class represents a generic queuing discipline that can be extended to implement different queuing algorithms, such as First-In-First-Out (FIFO) and Priority Queuing (PQ), among others.

The `QueueDisc` class defines a set of methods for enqueueing and dequeueing packets, computing queue statistics, and configuring various relevant queue parameters. Some of the key methods of the `QueueDisc` class include:

1. `Enqueue`: This method is used to add a packet to the end of the queue; conditions can be applied to this method to selectively enqueue packets according to a given algorithm (and drop packets which are not enqueued).

2. `Dequeue`: This method is used to remove the next packet from the front of the queue; similar to the `enqueue` method, conditions can be applied in this method to select which packets are dequeued, in what order, according to what priority, and more.

3. `SetMaxSize`: This method sets the maximum size of the queue, which determines how many packets can be stored in the queue at any given time. This method

enabled queue configuration in our simulated network environment.

4. `GetStats`: This method returns various statistics about the queue at the end of the simulation time, such as the number of packets in the queue, the average queue delay, and the queue drop rate.

Overall, the `QueueDisc` class provides a flexible and extensible framework for implementing packet queuing and scheduling algorithms in NS–3, and was a central component of this project. Conveniently, NS–3 also implements several common AQMs, which can be installed on a given `Node`'s `QueueDisc` to more accurately simulate the packet drop behavior at internet switches.

Most importantly, the generic `QueueDisc` class was extended to implement a custom policy, in which our RL agent continually modified the queue max size and accordingly dropped or enqueued incoming packets. We refer to this `QueueDisc` algorithm as `QueueDisc::RLAgent`. Details of this implementation are discussed later, in Section 4.3.

4.1.2 Traffic Generation: Bulk Send Application

In NS–3, `Applications` are software modules that run on network `Nodes` and generate traffic in the simulated network. Applications can be used to simulate various types of network traffic, such as web browsing, file transfer, and video streaming, among others.

The `ns3::BulkSendApplication` is a specific type of application in NS–3 that generates traffic by sending bulk data over a TCP connection, all at once. `BulkSend` works by creating a TCP connection to a remote node and immediately sending the specified number of bytes (`MAX_BYTES`) across that connection, waiting until all bytes are received (or until the simulation ends) to close the connection. There are several configuration options which specify the behavior of the application, such as the amount of data to send, the rate at which data is sent, and the destination address of the TCP connection, among others; carefully selecting our configuration allowed us to accurately simulate a congested network environment, with traffic that mimicked web search traffic (this is elaborated further in Section 4.2.2).

4.1.3 Network Events: Flow Completion, Packet Drops

NS–3 provides a powerful and highly customizable tracing and event notification framework which enabled us to monitor and analyze the behavior of our constructed simulation. Some of the key network events and characterizations which were relevant in our development process were packet transmissions, receptions, & drops, flow completions, and continuous queue lengths, among others. NS–3's tracing and notification framework allowed us to detect and report on these in real-time during our simulations, which was crucial in validating our topology and developing a clear picture of network performance.

Tracing functionality is provided by the NS–3 `Simulator` class via functions such as `Simulator::ScheduleWithContext()` and through NS–3 `Object` classes via functions such as `QueueDisc::TraceConnectWithoutContext()`. NS–3 also offers an event notification framework in the `ns3::EventId` class, which registers event handlers that are executed whenever a particular event occurs in the simulation. These tools were critical in developing much of our simulation data reporting:

- **Packet Drop** Packet drops were reported by the relevant `QueueDisc` when its installed AQM elected to drop an incoming packet, rather than enqueueing it.
- **Flow Completions** Flow completions were reported by the relevant `QueueDisc` when it had received/transmitted all the packets in a specified flow. This triggered an event handler which aggregated and reported flow data (size, throughput, completion time, total packet drops).
- **Throughput** Network endpoints were continuously monitored as incoming packets were accepted, with a scheduled throughput calculation occurring at a regular, specified time interval (usually 100ns).

The tracing and event notification framework in NS–3 allowed us to monitor the behavior of our simulated network in real-time. This served a dual purpose: during development, it allowed us to diagnose and troubleshoot network problems and validate our topology (discussed further in Section 4.4). Then, during simulation and data collection, it allowed us to generate useful, relevant data and prepare it for post-processing and analysis (this analysis is presented in Chapter 5: Results).

4.2 Simulation Environment

Having discussed the most relevant NS–3 components, we now specify the topology which was constructed for this research, and discuss the simulation environment.

4.2.1 Network Topology

In order to appropriately assess the performance of an RL agent in managing the buffer, it was also necessary to develop and understand a "baseline" buffer management scheme. To this end, two concurrent simulation models were developed: one which ran continuously with an unmodified ("static") buffer, and one which included an RL agent in the buffer management loop. The topology and network configuration were the exact same between these two simulation models to enable accurate and reasonable comparison between them; this shared topology is illustrated in Figure 5.

This topology consists of 3 main node classes: one sink node, one intermediate (congestion) node, and ten sender (source) nodes. The intermediary node acted as the point of congestion on this network, on which a `QueueDisc` (queue discipline) was installed. This node was configured to operate with the desired buffer management

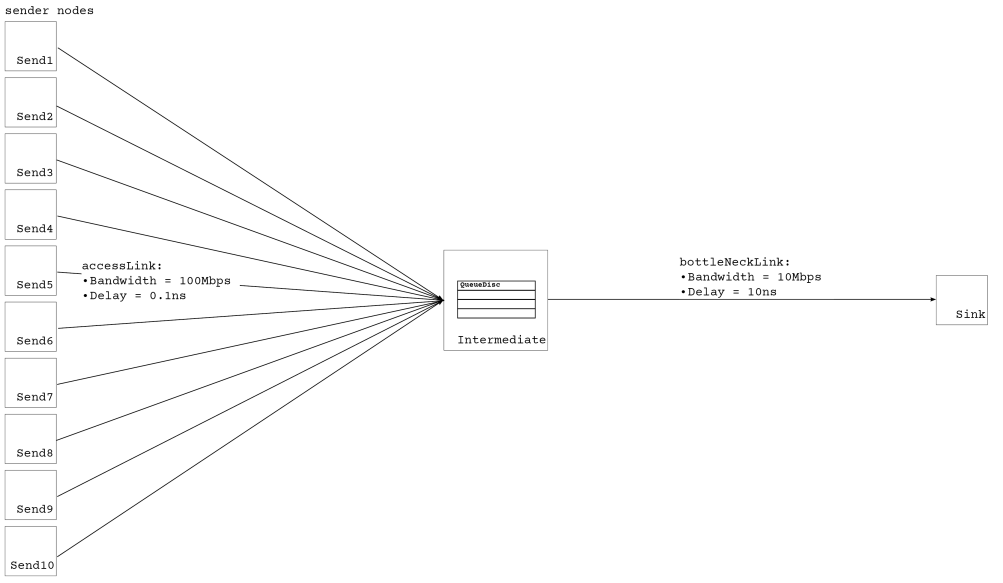

Figure 5: This was the basic simulation topology, used in both the Static Buffer and RL Buffer simulations. 10 sender nodes are connected via a high-bandwidth low-latency channel (`accessLink`) to the intermediate node, on which a `QueueDisc` is installed and configured as desired. The intermediate node is connected via a low-bandwidth high-latency channel (`bottleneckLink`) to the destination (`sink`) node, creating congestion at the bottleneck link.

strategy (static-size buffer or RL-agent buffer); this single queue handled all incoming packets from the sender nodes.

Sender nodes were configured with the desired TCP protocol[6], selected from `TCP` $\in$ {`TcpCubic`, `TcpNewReno`}. The choice of TCP impacted the implementation details of each sender's congestion control, resulting in different traffic send patterns.

The point-to-point connection between the sender nodes and intermediate node was the `accessLink`, because of its relatively high bandwidth (100Mbps) and low-delay (0.1ms) configuration. The point-to-point connection between the intermediate node and the sink node was the `bottleneckLink`, because of its relatively lower bandwidth (10Mbps) and high-delay (10ms) configuration. Packet transmission from the source node was limited by the bottleneck link in this configuration (100Mbps $\rightarrow$ 10Mbps bandwidth reduction at bottleneck), so packets waited at the intermediate node before being transmitted to the sink.

The intermediate node's queue was regulated by the queue discipline (`QueueDisc`) installed on its `accessLink NetDevice`. The `QueueDisc` was configured with the following parameters:

[6]Several TCP variations were available for sender configuration, including {`TcpBic`, `TcpHtcp`, `TcpHighSpeed`, `TcpHybla`, `TcpScalable`, `TcpWestwood`, `TcpYeah`}. These TCP versions are implemented in the NS–3 simulation library.

- `DiscType`, which assigns the Buffer Management (BM) & Active Queue Management (AQM) strategy[7] for this `QueueDisc`. For the static-size simulations, a basic `FIFO Droptail` queue was used, which enqueues and dequeues packets in the order in which they arrive, and accepts packets until it is full, at which point incoming packets are dropped; the buffer in these simulations was shared according to per-queue priority weights. In the RL simulation, the RL agent manages the buffer/queue thresholds.

- `MaxSize`, which assigns the maximum size of the queue in packets. For one set of simulations, the maximum queue size in packets was iteratively set $\texttt{MaxSize}_i \in \{100, 200, 300, \ldots, 1000\}$.

And lastly, each queue (Priority 0 Queue, Priority 1 Queue) was assigned some α according to its traffic priority. This alpha was used to weight certain types of traffic more heavily than others, i.e. by guaranteeing more buffer space for a queue with Priority 0 than for a queue with Priority 1. The αs used in each simulation are summarized in Table 1 below.

Simulation Type	(α_0, α_1)
Static Buffer: Naive	$(1, 1)$
Static Buffer: Sophisticated	$(8, 1)$
Reinforcement Learning Buffer	*

**The RL agent selects the α for each queue; this is discussed further in Section 4.3.2.*

Table 1: Per-queue α priority weights for each simulation type.

As implied by the name, the Static Buffer: Naive configuration equally shares the buffer between traffic of all priorities, potentially allowing lower-priority traffic (like long flows) to starve out higher-priority traffic; this will serve as our baseline configuration; we aim to outperform this buffer management strategy using RLBM. The Static Buffer: Sophisticated configuration strongly favors Priority 0 traffic (allocating 8x the available buffer for Priority 0 as for Priority 1 traffic). Lastly, in the Reinforcement Learning Buffer, the RL Agent selects the appropriate α values for each queue according to the policy it has learned; these α weights are manipulated frequently (every 0.1 seconds, for example) throughout the simulation time, as the RL agent learns about the environment (for more, see Section 4.3).

The constructed topology was simple, but offered several points of control which enabled us to simulate many network environments (i.e. multiple configurations of the same topology, varying TCP, network load, and queue/buffer size). Maintaining

[7] NS–3 enables several `DiscTypes`, including `PfifoFast`, `RED`, `CoDel`, `FqCoDel`, and `PIE`. In order to establish a controlled, baseline performance benchmark for the RL agent, we opted to use a very simple BM/AQM strategy.

a simple network topology also enabled us to more precisely compare simulation data between different configurations.

4.2.2 Traffic Generation

As discussed in Section 4.1.2, the senders in the specified topology ran `BulkSendApplications` to send large amounts of data to the sink node. Multiple `BulkSends` were installed on each sender node; the start time of each `BulkSend` was specified by generating a time delta t on a Poisson distribution over the average send rate. For each `BulkSend` that was installed, the start time for that flow was t seconds after the prior `BulkSend` app installed on that sender.

Furthermore, in order to most accurately simulate an Internet network environment, the size of each `BulkSend` was specified by generating a send size s according to a websearch-size CDF (cumulative distribution function) [2]. This enabled us to produce a realistic traffic profile in each simulation. A code summary of this traffic generation is specified below:

```
for node in senders:
   start_time = 0
   while start_time < SIMULATION_END_TIME:
      s = rand_CDF(websearch traffic size)
      BulkSendApp send = new BulkSendApp(SendSize = s)
      t = poisson(request_rate)
      start_time += t
      node.Install(send, StartTime(start_time))
```

Moreover, the load in each simulation was varied to analyze network behavior under varying amounts of traffic and congestion. As such, the request rate (`request_rate`) in each simulation was calculated according to `request_rate` $=$ `load` $* 500$ requests/s, where `load` was varied in $\{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$.

In order to test our buffer management scheme with multiple queues, traffic was assigned priorities according to the flow size; short flows were given Priority 0 (highest priority), whereas long flows were given Priority 1 (lower priority). This short/long flow distinction was determined according to size s: flows with $s < 2000$ packets were given Priority 0, whereas flows with $s \geq 2000$ packets were given Priority 1.

4.3 Reinforcement Learning Agent

As discussed in Section 3.2, a reinforcement learning algorithm was developed which was compatible with the existing Static Buffer simulation; the RL agent was turned on during selected simulations to manage the buffer at the intermediate node[8]. The RL

[8]This portion of the project builds on the work done by Haiyue Ma and Hengrui Zhang under Professor Maria Apostolaki [6].

agent operates as an on-the-loop, real-time buffer manager; its role in the simulation is illustrated in Figure 6 below.

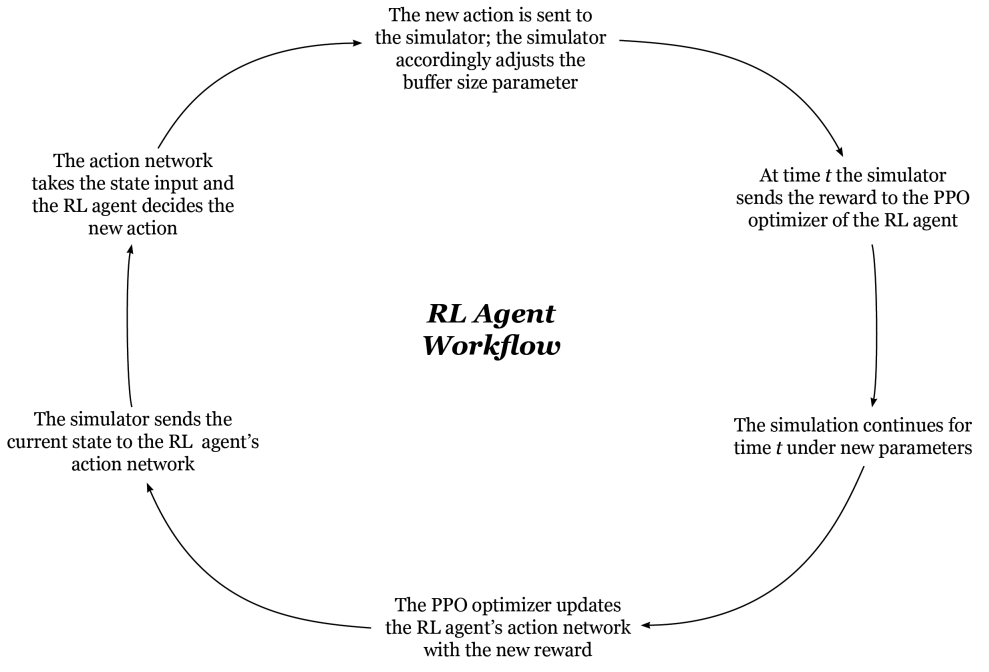

Figure 6: This workflow diagram illustrates the mechanism of interaction between the NS–3 simulation and the RL Agent. The simulator communicates state and reward information to the RL Agent, which in turn communicates actions to the simulator. These actions are buffer sizes, which the simulation can enact in real time by changing its packet drop policy.

The RL agent was implemented using the PyTorch C++ API; this presented a significant challenge in development, as PyTorch is a Python-based library that needs to be installed with specific dependencies that are not natively supported by the NS–3 simulation environment. In order to integrate PyTorch with the NS–3 `waf` build system, it was necessary to create a Docker container preloaded with `libtorch` and other relevant libraries and dependencies—this provided a consistent and reliable environment for the RL algorithm to run within the NS–3 simulation. This Docker image was used to run all simulations with the RL agent.

4.3.1 State Space

In designing the state space of the RL agent, it was important to select parameters which are available in real-world networks (to address some potential shortcomings of DT and ABM), and which comprehensively represent the state of the network traffic

and buffer (to ensure the policy addresses the total network environment). To this end, the parameters in Table 2 were selected as members of the RL agent's state space.

State Parameter	Description
β	Per-priority buffer allocation weight.*
`drain_rate`	Buffer drain rate.
Remaining buffer	Fraction of buffer which is currently empty.
Flow finish rate	Number of flows which have completed during this interval.
Packet loss	Number of packets which were dropped during this interval.

**Each queue (Priority 0, 1) was assigned some predetermined buffer allocation weight.*

Table 2: State Space Design for the RL Agent.

The state space was updated every time a flow completed in the network, to ensure the RL agent continually had the most up-to-date information from the simulation environment when it was "woken up".

The per-queue priority weight β was assigned at configuration time in each simulation; the RL agent was run with two configurations on β, summarized in Table 3. For the RL Buffer: No β scheme, P0 and P1 queues were assigned the same priority, such that their thresholds were determined solely by the RL-selected αs; this is effectively the same as assigning no β-weights. For the RL Buffer scheme, P0 was assigned $\beta = 1$ and P1 was assigned $\beta = 0.5$; this can be viewed as "guaranteeing" Priority 0 traffic twice as much of the remaining buffer space as Priority 1 traffic (though the RL Agent may select α values which outweigh the β P0 bias). This β is implemented to "aid" the RL agent in the earliest rounds of threshold selection at the beginning of the simulation, when all or most of the state/action space is still unexplored, with the goal of converging upon optimal thresholds more quickly.

Simulation Type	(β_0, β_1)	
RL Buffer: No β	$(1, 1)$	*Equal Weight*
RL Buffer	$(1, 0.5)$	

Table 3: Per-queue β weights assigned in each RL simulation type.

4.3.2 Action Space

The RL agent was responsible for setting the per-queue threshold in the buffer, as discussed in Table 4; this represents a continuous positive action space with a practical upper bound given by the physical maximum size of the buffer.

In the simulations, the "physical" size of the buffer is determined during the topology construction, and is varied on a per-simulation basis. Then, the RL agent is able to select

Action	Description
Buffer Per-Queue Threshold, α	Values in $\{0, 1\}$, where 1 represents 100% of the physical size of the buffer.

Table 4: Action Space Design for the RL Agent.

how much of the given buffer is available to incoming traffic according to its priority: for example, although the buffer in the constructed topology may have capacity for 1,000 packets, the RL agent can choose to restrict the buffer to 200 packets for traffic with Priority 1 (*action* $\alpha = 0.2$). Then, if the buffer is already storing 200 Priority 1 packets, any incoming Priority 1 packets will be dropped until more buffer is available (e.g. the buffer stores 199 or fewer Priority 1 packets, or the RL agent increases the buffer max size for Priority 1 traffic, like *action* $\alpha = 0.4$).

Then, given some action α decided by the RL agent, the per-queue threshold in the buffer is determined according to the following function:

$$\text{Queue\_MaxBuf} = \alpha * \beta * B_a,$$

where B_a denotes the available space in the buffer and β is the predetermined per-queue weight discussed in Table 3. In the RL Buffer: No β scheme, this threshold equation is reduced to

$$\text{Queue\_MaxBuf} = \alpha * B_a$$

4.3.3 Reward Function

The reward for the RL agent was based on two indicators of network performance: FCT and throughput. These are two critical metrics in buffer management, as they can indicate if the buffer is too large[9]:

- *Long flow completion time* can indicate that packets are getting "stuck" in the buffer, which is not draining at a reasonable rate given how many packets it currently stores. This means that the buffer has grown too large, and is introducing too much delay for flows that are ongoing.
- *Low throughput* indicates that the buffer has become a bottleneck point in the network, with packets similarly suffering from excessive queueing delay. This once again indicates that the buffer has grown too large.

Whereas throughput presents a generalized picture of current network performance (across all flows and network traffic), worst FCT slowdown specifically reports on the single worst outcome in the network currently, and can take on more drastic values (to

[9] *Note:* as a reminder, the "optimal" buffer size is the minimum size which results in maximum performance. In other words, the buffer should be as large as required by the network traffic, but no larger than required.

indicate times when the RL agent should respond more strongly, reducing the buffer size by a large amount). These parameters are discussed in Table 5.

Reward Parameter	Description
Worst FCT slowdown	The worst FCT slowdown among all flows in this interval.
Throughput	Number of packets received in this interval ÷ time delta.

Table 5: Reward Function Design for the RL Agent.

FCT slowdown is computed by subtracting the ideal FCT[10] from the actual FCT of a given flow. For some flow f,

$$\text{slowdown}(f) = \text{FCT}_{\text{actual}}(f) - \text{FCT}_{\text{ideal}}(f)$$

Then, the worst slowdown at time t_i with interval length 0.1 seconds is computed as

$$\text{WFS}(t_i) = \max(\text{slowdown}(f) \, \forall f \in \text{Completed Flows}(t_i - 0.1, t_i))$$

where Completed Flows$(t_i - 0.1, t_i)$ represents all flows which have completed in the past 0.1 seconds (one interval).

Throughput is computed by dividing the number of packets transmitted in a single time interval by the length of that time interval. The RL agent's reward is updated on a regular basis, at an interval set by the programmer (e.g. every 0.1 seconds). At the end of each interval, the number of packets received at the sink node *during that interval* is reported to the RL agent and the throughput is computed; for the next interval, the packet counter starts at 0. For intervals of length 0.1 seconds, at time t_i:

$$\text{throughput}(t_i) = \text{packet\_count}(t_i - 0.1, t_i)/0.1$$

Using these metrics, the reward function for the RL agent is computed as

$$\boxed{\text{reward}(t_i) = -\log(\text{WFS}(t_i)) + \log(1 + \text{throughput}(t_i))}$$

Although the state is updated every time a flow completes, the reward is only updated at the specified interval.

Given our discussion of the state space, action space, and reward function of the RL agent, we summarize the interaction between our NS–3 simulation and the RL agent[11] in Figure 7.

[10]The *ideal FCT* is how long a given flow would take to complete if it stood alone in the network; this is determined by the bottleneck bandwidth in the network, the link round-trip-time (RTT), and the flow size.

[11]This illustration of the total system interaction was created by Haiyue Ma and Hengrui Zhang.

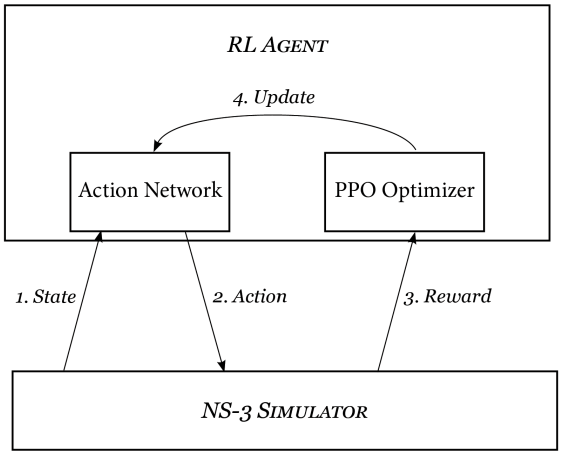

Figure 7: Summary illustration of the interaction between the RL agent and the NS–3 simulation.

4.4 Network Validation & Analysis

The first objective in developing our topology was to validate that it was working as expected. By default, the NS–3 simulation environment provides little insight about the progress of the simulation, the flows being transmitted over the network, and the state of each component during the simulation. As a result, it was necessary to develop a data collection workflow for each simulation (real-time and post-simulation) and establish several benchmarks to validate the topology, network traffic, and network components.

In the first round of data collection a "basic" configuration was assigned to the simulation as follows: a single priority queue of fixed size (100 packets), a fixed buffer size (100 packets), and static queue control policy (FIFO DropTail). All senders ran the TCP NewReno protocol, and sent 100,000 bytes (1,000 packets) one time over the network to the sink node. In this run, data was collected on the network's performance metrics, such as throughput, delay, and packet loss, as well as the buffer usage over time. This data was then analyzed to ensure that the network was performing as expected and that the topology was configured correctly.

4.4.1 Runtime Metrics

These performance metrics were reported during the simulation, and used both during and after the run to validate our network configuration and simulation.

- **Throughput** The throughput at the bottleneck was reported at regular intervals during the run (every 100ns), to observe throughput changes in response to changes in traffic. It was important to observe that, during peak packet transmission, the throughput at the bottleneck link was no greater than 10Mbps, as set in the simulation code. This gave rise to a "congested" condition in the network, as the access link exhibited throughput up to 100Mbps.

- **RTT (Round-Trip Time)** The round-trip time of packets (from sender to sink node) was determined in real-time by installing a `PingApp` on one sender node. The `PingApp` sent a single packet once every second (in simulation time) and waited for the ACK response; upon receiving the ACK response, it reported the time delta from packet send to ACK. This time delta is the RTT, which reflects the actual time it takes for a packet to travel across the network, accounting for any delays introduced by network congestion, packet loss, and other factors. It was important to observe that, when there was no congestion, the RTT was 10 ns, as set in the simulation code.

- **Queue Length** The length of the bottleneck queue was recorded at each packet enqueue/dequeue event, to provide a detailed understanding of the queue behavior. Additionally, it was important to observe that the queue length never exceeded the size set by `QueueDisc::SetMaxSize()` in the simulation code.

- **Packet Drops** Packet drop events were reported to provide insight into the behavior of the queue at the bottleneck link. We observed no packet drop events when the traffic load was light (no congestion), with a corresponding increase in packet drop events as the network load was increased. The total number, frequency, and timing of packet drops also varied with the choice of AQM on the intermediate node's `QueueDisc`, as expected.

- **Flow Completion** Flow completion time was reported for all flows in the network (`BulkSendApplications` on each sender). When flow size was fixed and the traffic load was light (no congestion), all flow completion times were roughly equal, as expected (no delay was introduced at any point in the network, and flows of the same size exhibited the same time to completion).

4.4.2 Retrospective Metrics

These performance metrics were reported at the end of the simulation, and used in post-processing to validate our network configuration and simulation.

- **Flow Monitor Data** NS–3 provides a `FlowMonitor` class as a built-in monitoring tool, which observes all flows across the network and reports detailed flow-level statistics such as packet counts, byte counts, start and end times, and flow duration at the end of the simulation. This data provides retrospective insight into the performance of the network configuration for a given simulation, and allowed us to observe changes in the behavior of network traffic as we varied simulation parameters.

- **Total Throughput** The total throughput was also reported at the end of the simulation: this represented the total number of packets received at the sink node divided by the total simulation time (60 seconds), and varied slightly from

the intra-simulation throughput reporting, as it provided a total picture of the throughput (including times of low congestion, where perhaps no packets were being transmitted).

- **Total Packets Received** The total number of packets received at the sink node was used to calculate the total throughput, and also indicated whether all data sent by the sender nodes had actually been received. In simulations with high network load, we observed that only a fraction of sent packets arrived at the destination, which confirmed our expectation that the network was experiencing congestion for the full duration of the simulation. This was useful in validating our traffic generation policy.

4.4.3 Expectations for the Data

Each of the metrics described above was associated with a set of expectations which, if met, would show that our simulation was working as designed. After verifying, updating, and finally validating our "basic simulation" against these metrics, we were able to implement changes to the topology which increased our simulation's state space, using multiple TCP protocols, BM/AQM algorithms, queue and buffer sizes, and traffic generation patterns. Using the data collection and analysis pipelines that were built during the validation phase of the project, the simulation was run again in numerous configurations, including with an RL–based buffer management algorithm. The data collected during these simulation runs was then analyzed to validate the performance of the RL–based buffer management algorithm and compare it to the results from the optimal fixed buffer size simulation. These simulation comparisons are presented in detail in Chapter 5: Results.

5 Results

We will now present the most significant results from our simulations. The full set of plots and data can be found in Appendices B & C.

5.1 Scope

This thesis work focuses on a single, fixed simulation topology in order to compare the Reinforcement Learning Buffer Management scheme to the Static Buffer baseline. As such, these results do not generalize to all possible topologies and network configurations, but rather motivate the exploration of RL in buffer management solutions. We do not extrapolate from these results to possible outcomes in real-world implementations of RLBM.

Even within our limited scope, it was necessary to configure our network and simulation parameters so as to enable us to draw significant conclusions. In order to better generalize these results within our topological scope, each set of simulations was run with a different random seed, which determined the randomly-generated `BulkSendApp` send size and start time. For each specified network configuration S(BM, TCP, Load, MaxSize), the random seed was fixed such that all possible combinations of parameters of S would be running the same simulation (i.e. the same exact traffic pattern). Then, the data across all random seeds was averaged to produce a comprehensive picture of the performance outcomes of each configuration.

5.2 Per-Simulation Configuration

To summarize, the parameter space of each simulation S(BM, TCP, MaxSize, Load) was:

- BM algorithm $\in$ {SB: Naive, SB: Sophisticated, RLBM, RLBM: No β},
- TCP CCA $\in$ {TCP CUBIC, TCP NewReno},
- Queue Max Size $\in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ packets,
- Buffer Max Size = Queue Max Size $*$ # Queues, and
- Network Load $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\} * 500$ requests/s.

For each simulation type, a specific set of per-priority buffer threshold allocation weights was either predetermined (in simulation configuration), or assigned by the BM scheme selected (i.e. the RL Agent). Table 6 presents a summary of these α and β threshold coefficients.

Simulation Type	(α_0, α_1)	(β_0, β_1)	
Static Buffer: Naive	$(1, 1)$	—	
Static Buffer: Sophisticated	$(8, 1)$	—	
RL Buffer: No β	*	$(1, 1)$	*Equal Weight*
RL Buffer	*	$(1, 0.5)$	

**The RL agent selects the α for each queue.*

Table 6: Per-queue α and β weights assigned in each Buffer Management configuration.

5.3 Statically-Configured Buffer

In order to understand the RL agent's performance outcomes, we first ran simulations with a statically configured buffer. This configuration was as follows:

- Queue MaxSize $\in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ packets
- Physical Buffer MaxSize = Queue MaxSize $*$ Number of Queues
- No AQMs

As per Table 6, the per-queue thresholds in the static buffer were determined according to pre-determined alphas, which weighted each queue according to its priority. We completed our simulations with two different sets of alphas: for the first configuration, which we refer to as Static Buffer: Naive, all queues were given the same priority. For the second set, Priority 0 traffic was assigned $\alpha = 8$, whereas Priority 1 traffic was assigned $\alpha = 1$ (effectively weighting P0 traffic 8 times higher than P1); we refer to this configuration as Static Buffer: Sophisticated.

5.4 RL-Managed Buffer

In order to draw valid comparisons between the RL-managed buffer and the statically-configured buffers, we maintained the same simulation topology and varied the same parameters; in other words, our configuration was as follows:

- Queue MaxSize $\in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ packets
- Physical Buffer MaxSize = Queue MaxSize $*$ Number of Queues
- No AQM; RLBM selects per-queue buffer thresholds

Furthermore, as per Table 6, we fixed two sets of β values for the RL Agent. In the first, Priority 0 traffic was weighted twice as much as Priority 1 traffic, in the scheme we refer to as RLBM. In the second set, , which we refer to as RLBM: No Beta, the β weights for both Priority 0 and 1 were $\beta = 1$.

5.5 Performance Comparisons

We will now discuss observed performance outcomes, comparing the RL buffer management schemes to the Static Buffer schemes. In this section, our key performance metrics are throughput and 99th percentile FCT slowdown, as these are members of the RL Agent's reward function, $\text{reward}(t_i) = -\log(\text{WFS}(t_i)) + \log(1 + \text{throughput}(t_i))$.

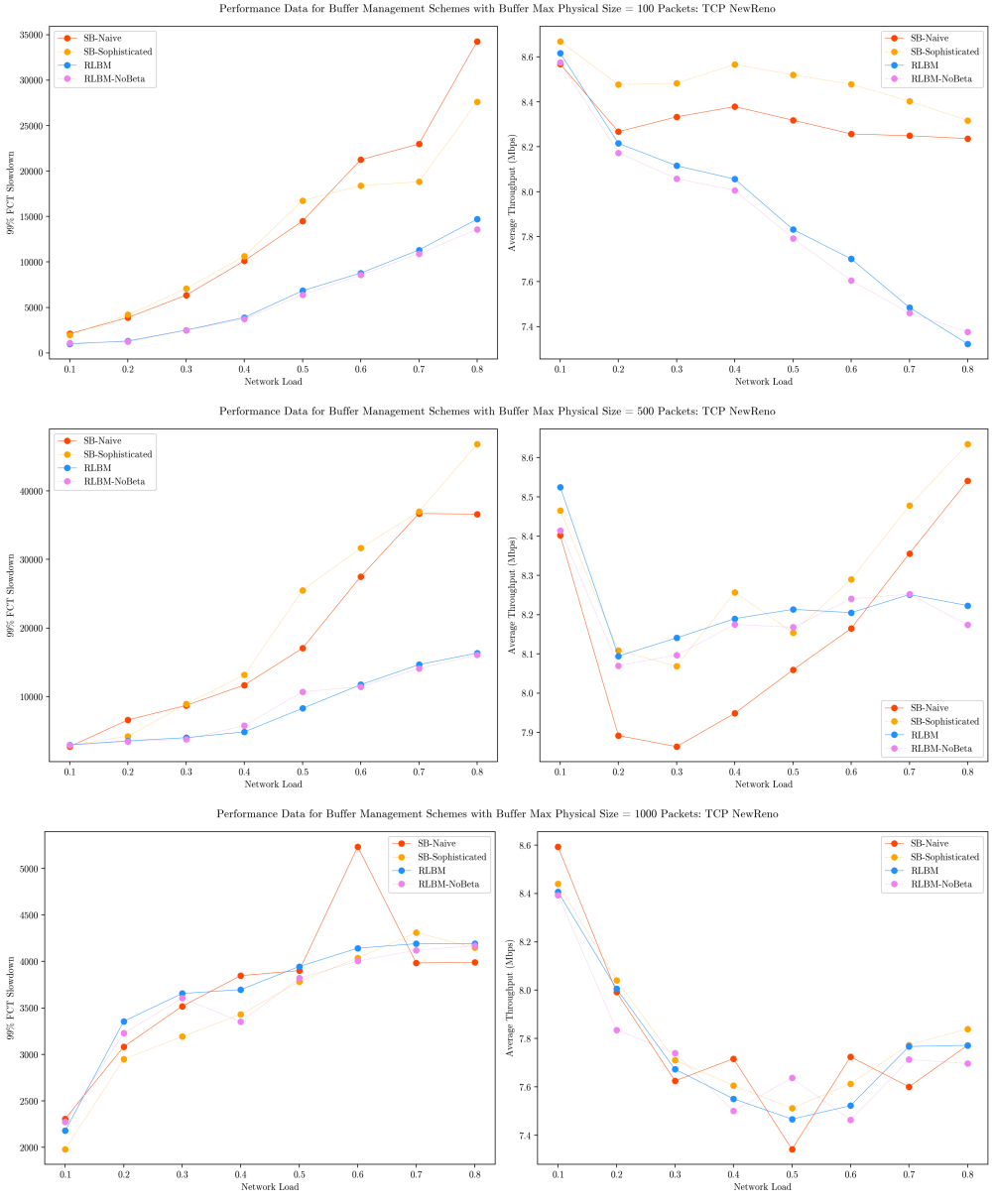

Figure 8: Throughput and FCT slowdown comparisons between Static Buffer and RL Buffer Management schemes, with different physical buffer maximum sizes (100, 500, 1000 packets). Each data point represents the average of 10 randomly-seeded simulations under the specified configuration.

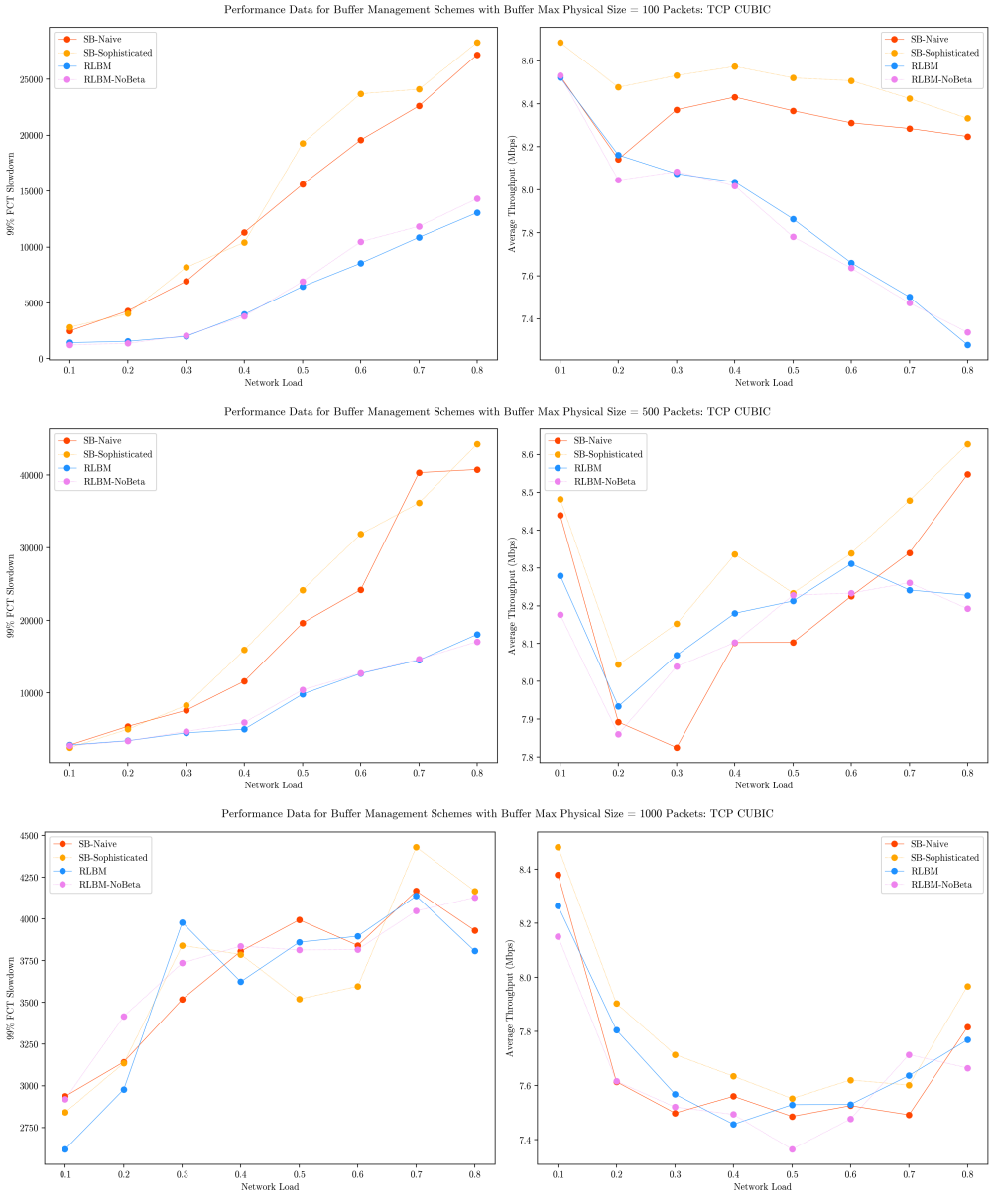

Figure 9: Throughput and FCT slowdown comparisons between Static Buffer and RL Buffer Management schemes, with different physical buffer maximum sizes (100, 500, 1000 packets). Each data point represents the average of 10 randomly-seeded simulations under the specified configuration.

There are some key patterns which emerge in this performance data: in Figures 8 and 9, we observe significant throughput degradation for the RLBM scheme, as compared to SB (both Naive and Sophisticated) with small physical buffer maximum size (100 packets). When the buffer's physical size is increased to 500 packets, the RL Buffer begins to outperform the Naive Static Buffer, seeing the same or better throughput

under most network loads. With a buffer max size of 1,000 packets, the RL Agent manages the buffer with minimal throughput degradation compared to the Static Buffer (both Naive and Sophisticated).

As for the 99th Percentile FCT Slowdown, this can be approximately restated as the "worst observed FCT slowdown" in a given simulation. For small physical buffers, RLBM showed significant improvement in worst FCT slowdown, as compared to SB (both Naive and Sophisticated); in other words, all flows completed in a time more similar to the *ideal* flow completion time under RLBM than under SB.

In general, the RLBM scheme exhibits lower buffer utilization than the SB schemes. Whereas SB fully utilizes the buffer in times of congestion, RLBM selects α values per-queue which maintain some guaranteed amount of available buffer; as a result, we observe lower 99th Percentile Buffer Occupancy (see Figure 17); however, this would also result in better burst tolerance in networks with bursty traffic profiles, as RLBM effectively maintains an "burst-absorption" tolerance in the buffer. These observations follow this pattern for all simulated Buffer Max Physical Sizes, strengthening our observations. These results are plotted for all simulation configurations, with the addition of 99th Percentile Buffer Utilization. Refer to Supplemental Figures 17–22 for the full set of plots.

We also observed the average flow completion time for each buffer management algorithm, as well as the average number of completed flows across 10 randomly-seeded simulations. Our per-configuration results are plotted in Figure 10. The full set of plots can be found in Figures 23–26.

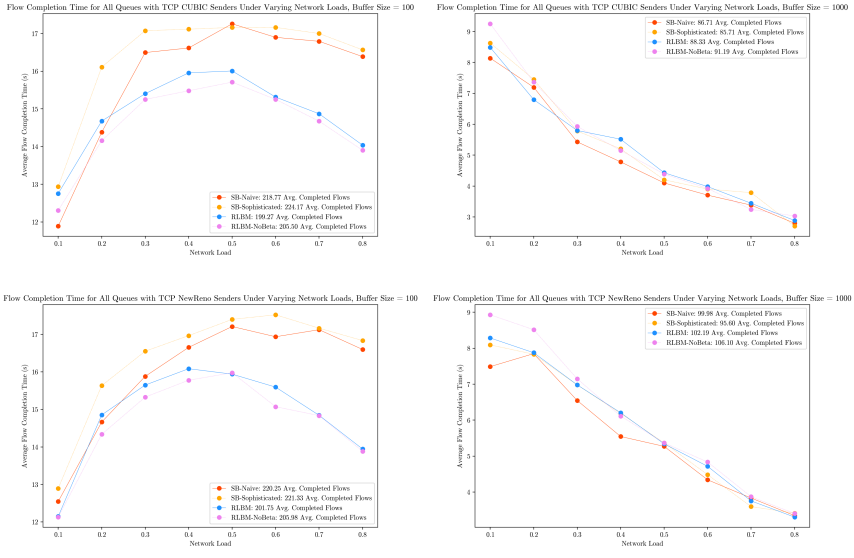

Figure 10: Average flow completion time of each BM strategy under varying Network Loads for Senders configured with TCP CUBIC and NewReno; Buffer Sizes in 100 and 1000 packets.

We see that, for small buffers, RLBM (in both configurations) produces shorter average FCT. This trend reverses for larger buffer sizes, where RLBM instead begins to result in more flows completed overall (in the same simulation duration). These results indicate slight improvement in FCT for RLBM over SB, as well as very stable average FCT (even with varying buffer sizes).

5.6 Buffer Comparisons

We will now turn to a discussion of our observations of the state of the buffer in the course of each simulation.

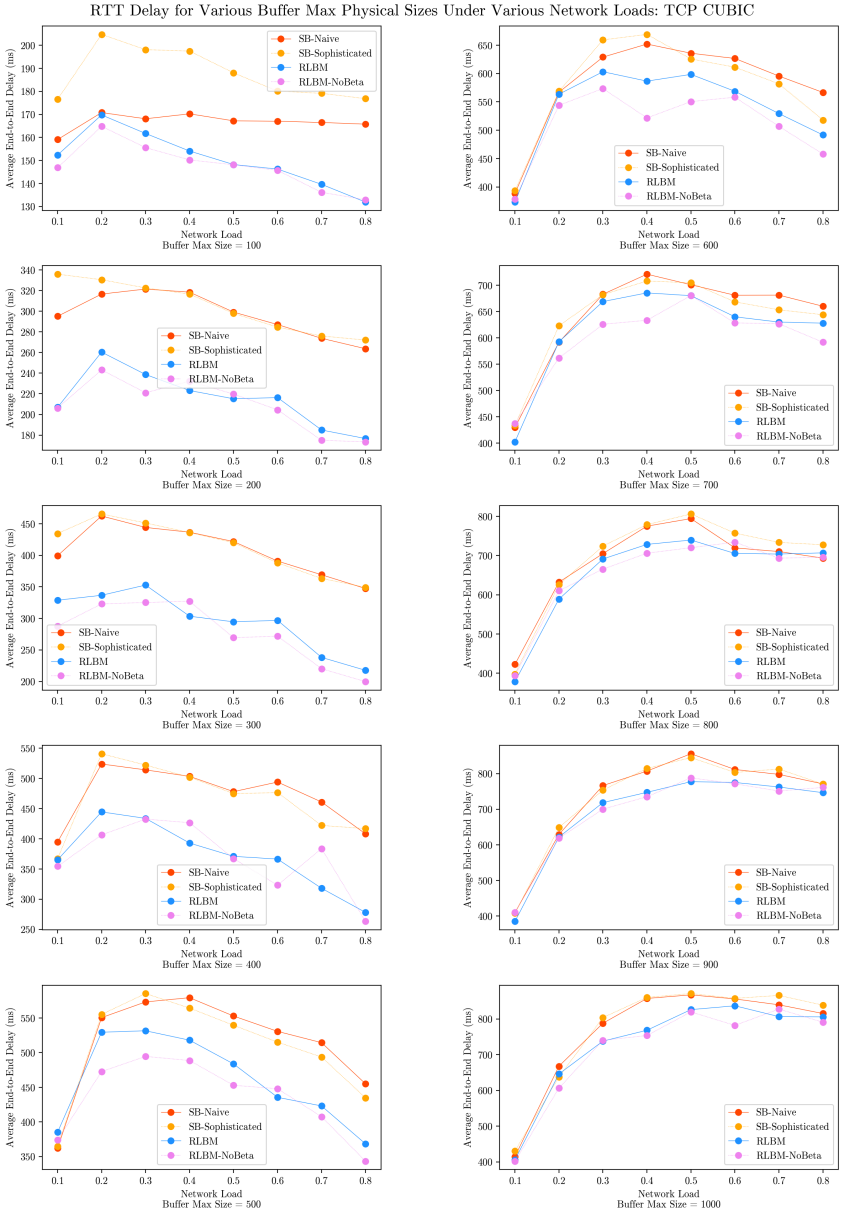

Figure 11: Observed end-to-end delay for ping packets in simulations with all senders configured with TCP CUBIC, and varying Buffer Sizes.

Figures 11 and 12 depict the average end-to-end delay experienced by ping packets (discussed in 4.4) in the course of each simulation. In general, all buffer management

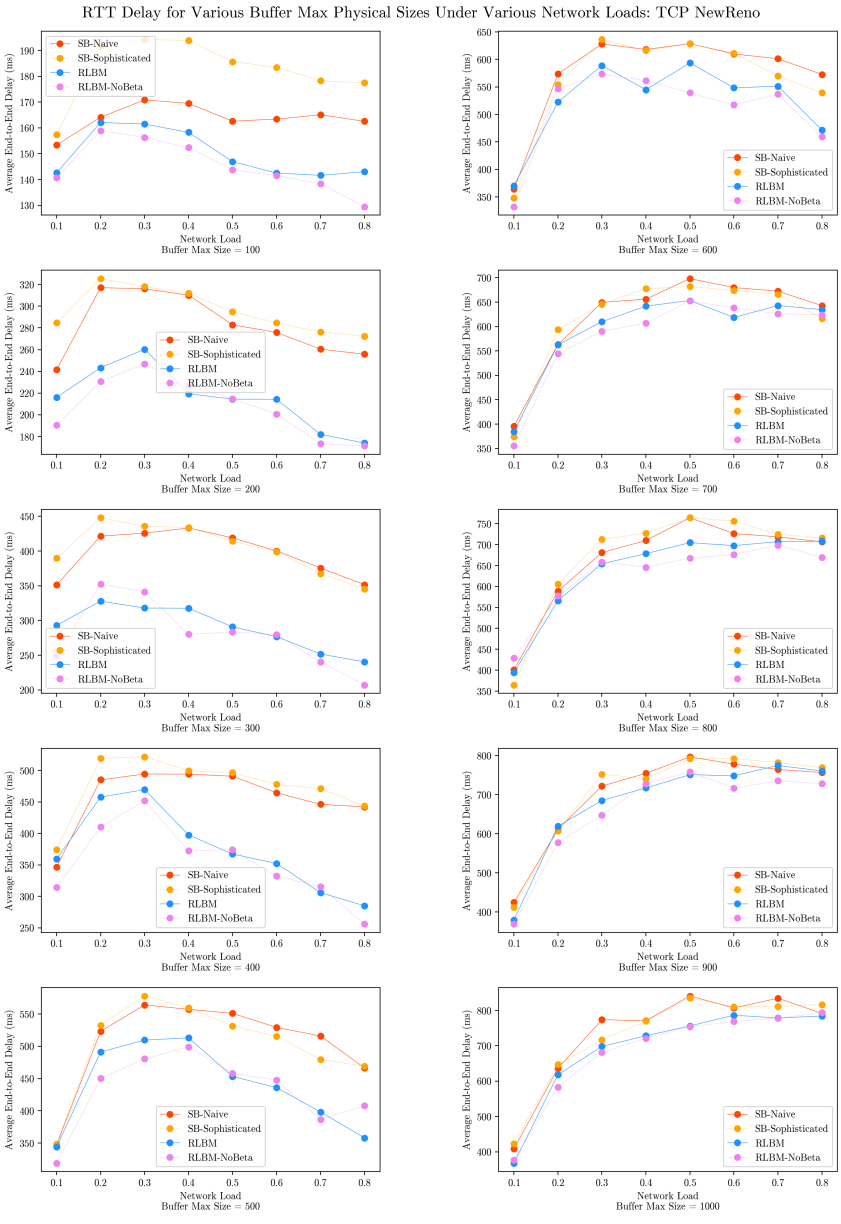

Figure 12: Observed end-to-end delay for ping packets in simulations with all senders configured with TCP NewReno, and varying Buffer Sizes.

schemes produce worse end-to-end delay for larger buffer sizes (this is the observation which gives rise to the "bufferbloat" problem). Furthermore, under higher network traffic loads, all BM schemes trend towards worse end-to-end delay, as expected. However, for smaller physical buffer max sizes, we observe significant improvements in end-to-end delay for the RLBM schemes, in both the TCP CUBIC and TCP NewReno sender

configurations. For large physical buffer max sizes, RLBM still produces lower end-to-end delay than the SB schemes, but to a less significant degree. In networks where end-to-end delay is a high-priority QoS metric, RLBM may outperform existing buffer management strategies.

Figures 13–15 plot the length of each queue (Priority 0, Priority 1) in the course of a given simulation

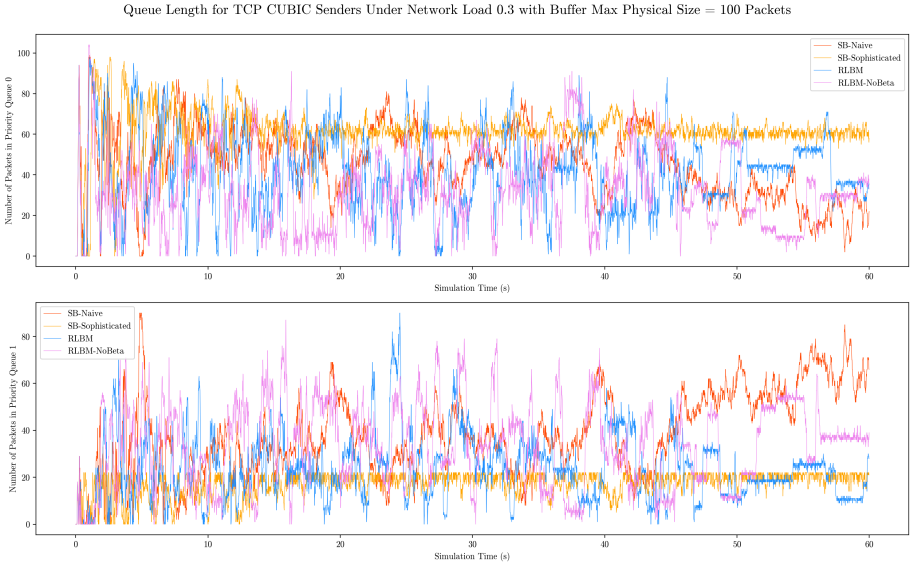

Figure 13: Queue length over the course of the simulation, configured as specified. We observe that the queue length never exceeds the physical maximum size (100 packets). Moreover, observe the impact of the fixed per-queue α weights in the Static Buffer Schemes, with the length of the Priority 1 queue being practically limited by the buffer capacity and the 8x α weight of the Priority 0 queue.

For all buffer max sizes, we observe that, on average, the Static Buffer Management schemes (both Naive and Sophisticated) have longer queues than the RLBM schemes, particularly the RLBM scheme with β weights. This observation accords with our observation of end-to-end delay, as we expect shorter end-to-end delay when there is a shorter standing queue. Furthermore, these characteristics of the RLBM algorithm likely give rise to our earlier observation, that RLBM produces a shorter 99th Percentile FCT Slowdown as compared to SB; in general, with a shorter standing queue, a given flow will complete more quickly (its packets won't get stuck at the end of a long queue, and will instead be dropped and retransmitted). We expect to observe this effect, as the RL Agent's reward function drives it to minimize the worst FCT slowdown observed in a single time interval. Overall, these observations of the buffer state during the simulation illustrate the differences in each algorithm, and demonstrate the practical manifestation of the RL Agent's action–reward mapping in its choice of per-queue thresholds.

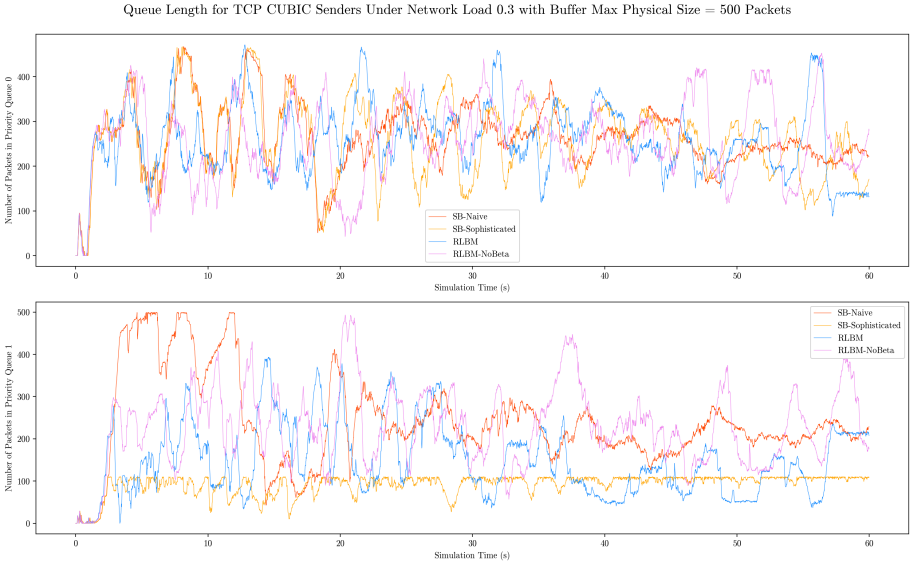

Figure 14: We begin to more clearly observe the difference between the RLBM scheme with pre-determined β weights and without; the RLBM: No β scheme allocates more buffer space to the Priority 1 queue than the RLBM scheme.

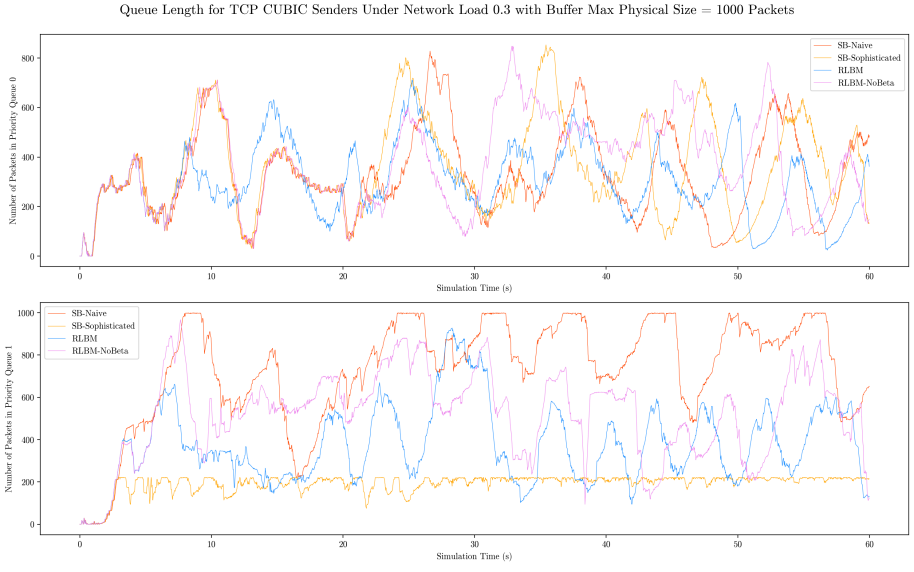

Figure 15: We can observe how the algorithms begin to diverge from one another after the first few seconds of simulation time. Both RLBM algorithms maintain, on average, a shorter queue than Naive SB. For the Priority 0 queue, both RLBM algorithms also maintain a shorter queue than Sophisticated SB.

5.7 Optimally Configuring the RL Agent

All data presented in Sections 5.5 and 5.6 was collected with an RL agent whose environment update rate[12] was set to 100000000 ns (0.1 s). In order to observe the impact of the update rate on the RL Agent's performance, we varied the `nanodelay` time interval of the RL loop in a smaller set of simulations (all senders configured with TCP CUBIC, on buffer physical max sizes in [100, 500, 1000]) and observed performance outcomes, depicted in Figure 16.

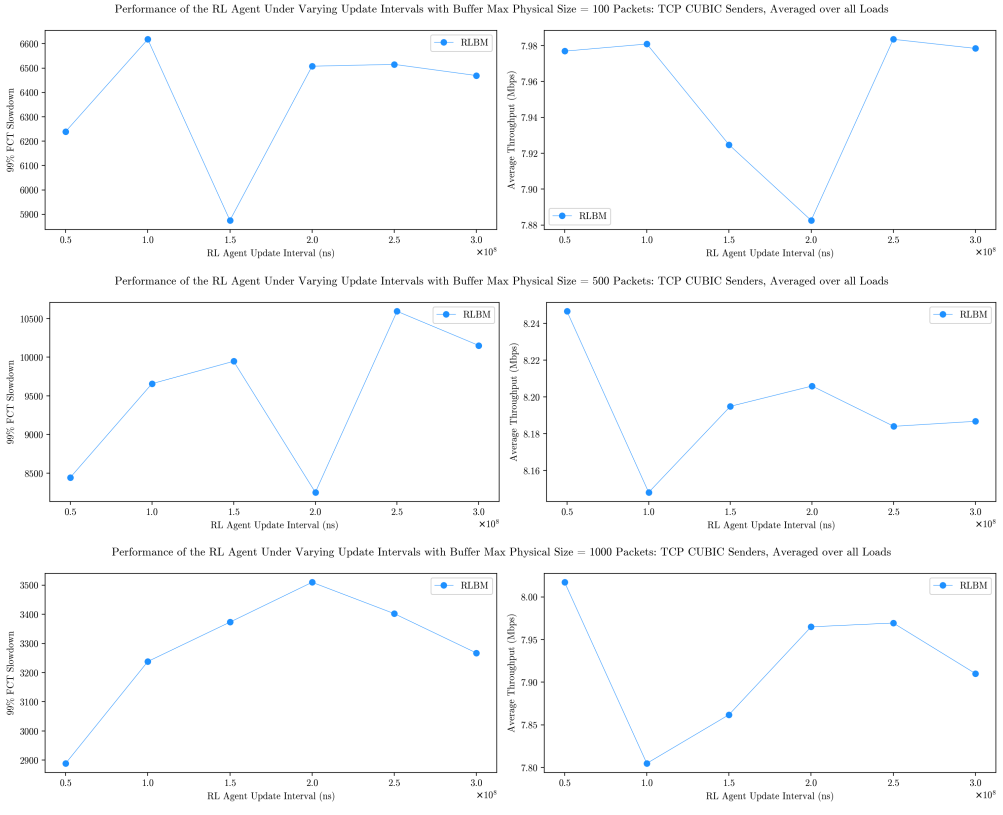

Figure 16: Performance outcomes of the RL Agent under varied configurations: 99th Percentile FCT Slowdown and Average Throughput of RLBM with varying `nanodelay` $\in [50000000, 100000000, 150000000, 200000000, 250000000, 300000000]$ ns. Each data point represents the performance outcome at a fixed `nanodelay`, averaged across all network loads (`load` $\in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]$)

We observe no distinct trends across all buffer max sizes. However, it is apparent that, for all buffer max sizes (100, 500, and 1,000 packets), a `nanodelay` of 50000000 ns (0.05s) results in the highest average throughput across all network traffic loads. This choice of `nanodelay` also results in minimal or near-minimal 99th Percentile FCT Slowdown. These results point to the use of 50000000 ns as the preferred `nanodelay` for

[12]Referred to as the agent's `nanodelay`, the update rate is the time interval at which the agent receives state and reward updates from the environment and determines a new action.

our constructed RL Agent, though more rigorous simulation is necessary to determine the optimal `nanodelay`.

Our conclusions are limited by the scope and parameter variation in the simulations conducted for this data; in order to form a more comprehensive, and perhaps a more conclusive, picture of the optimal RL Agent configuration, these simulations must be run under multiple buffer max sizes, TCP Senders, and with higher `nanodelay` granularity. Indeed, a wider range of possible `nanodelay` values would be useful in determining a theoretical and practical limit for the RL Agent's update rate in real-world network implementations as well.

5.8 Discussion

We observed improvements in RLBM's performance outcomes on buffers with larger physical sizes; this indicates that the RL Agent performs better when given a larger action space. Whereas RLBM produced worse throughput in small buffers, RLBM produced the same or better throughput as SB in larger buffers. Moreover, we observed that RLBM resulted in improved FCT over the SB scheme on average, with particularly strong improvement in buffers with smaller physical max sizes. This motivates further exploration of RLBM in buffers under various configurations (particularly buffers of varying physical size).

Furthermore, we observed the behavior of each algorithm in the buffer, noting several characteristics which indicated that our RL Agent was working as expected. These observations are important to validate the RLBM scheme and detect changes in the network (i.e. traffic pattern) and the ensuing response of the RL Agent. A key observation in the buffer behavior was that RLBM exhibited lower 99th Percentile Buffer Occupancy than SB; this could result in significatly improved burst tolerance, and warrants further exploration in network configurations with bursty traffic profiles.

Lastly, we began manipulating the parameters of the RL Agent itself, to understand how to optimally configure the RL algorithm. In practice, the `nanodelay` of the RLBM scheme is limited by the computational resources available to the RL Agent, the needs of the network, and the network profile (i.e. how dynamic a given network is). In our simulations comparing RLBM to SB, the `nanodelay` was fixed at 0.1s, which is a reasonable timescale for real-world applications—indeed, the observed performance improvements of RLBM over SB with `nanodelay` = 0.1s demonstrate the feasibility of an RL–based implementation in real-world buffer management schemes.

6 Future Work

This work explored, within a limited scope, the possibility of using reinforcement learning to manage the buffer at congested network nodes. Our findings, in the context of the greater body of work in RL–based networking solutions, warrant further exploration in this area. To build upon the work completed in this thesis, we first propose a local exploration of the RL Agent simulated in this work, to determine its optimal configuration; then, to extend the findings of this work and implement reinforcement learning in real-world networks, we propose the implementation and simulation of RLBM with various action spaces, and in many topologies.

6.1 Optimizing the RL Agent

An immediate focus of future work in this thesis project will be to utilize the validated NS–3 topology and data processing framework discussed in Chapter 4: Design & Implementation to find an optimal configuration for the RL Agent itself. As discussed in Section 4.3, the RL agent was given a particular state vector and reward function to learn a policy on the optimal buffer size for the topology and network traffic. Furthermore, the `nanodelay`, or update interval, of the state and reward given to the RL agent was fixed across all simulations. It is worth exploring multiple state vectors, reward functions, and `nanodelay` values to produce an RL Agent which is optimally configured to find the ideal buffer size in the given network configuration. In particular, we propose adjusting the reward function,

- *to include buffer utilization.* Our results show that the RLBM scheme underperforms SB in 99th Percentile Buffer Utilization. Including buffer utilization in the reward function would serve to incentivize the RL Agent to maximally utilize the buffer resources available to it; practically speaking, an implementation with higher buffer utilization makes better use of network resources and is therefore more attractive for real-world implementations.

- *to prioritize throughput more heavily.* In smaller buffers, we observed throughput degradation for the RLBM schemes as compared to the SB schemes. While this was not the case for larger buffers (where RLBM performed the same or better on average, compared to SB), we expect that RLBM could achieve higher overall throughput (across all buffer sizes) if the reward function more heavily weighted throughput. Adjusting the coefficients in the reward function may produce more compelling results in favor of RLBM.

6.2 Choice of RL Action Space

We chose to focus the scope of our work on buffer sizing, but there are many other variables in networking, both continuous and discrete, which could be manipulated by

a reinforcement learning algorithm. For example, if each queue in a given switch was configured with some AQM strategy, the RL Agent operating at the switch could select and modify the AQM at each queue, optimizing on similar performance metrics like throughput and FCT. It is worthwhile to pursue more constrained and less constrained action spaces, continuous and discrete action spaces, and generally explore the numerous variables in networked environments which can be controlled by a reinforcement learning algorithm.

6.3 Extending to Other Topologies

Ultimately, the goal of this work was to motivate further exploration into the application of reinforcement learning algorithms in buffer management. Though our topological scope was limited, the results warrant further exploration, particularly in more complex (and real-world) topologies. We aimed to simulate internet traffic in our topology, though RLBM may offer improvement in networks with other traffic profiles.

Our results in Section 5.7 indicate that a `nanodelay` of 50000000 ns is optimal for the established topology (though these results are not conclusive). Future simulations should be conducted with this parameter configured as such, in order to compare an optimally-configured RLBM scheme to other schemes; in general, determining the "optimal" configuration for the RL Agent will be a necessary step in implementing reinforcement learning algorithms in networks.

References

[1] *The NS-3 Network Simulator, http://www.nsnam.org.*

[2] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: Active Buffer Management in Datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM 2022, page 36–52. Association for Computing Machinery, 2022.

[3] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, aug 2004.

[4] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. Lfq: Online learning of per-flow queuing policies using deep reinforcement learning. In *2020 IEEE 45th Conference on Local Computer Networks (LCN)*, pages 417–420, 2020.

[5] Ivo Grondman, Lucian Busoniu, Gabriel A. D. Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[6] Haiyue Ma and Hengrui Zhang. Reinforcement learning for active buffer management.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Subir Varma. *Internet Congestion Control.* Elsevier Science, 2015.

A Engineering and Industrial Standards

The independent project described in this thesis incorporated the following engineering and industrial standards:

- International units of physical quantities:

  International System of Units (SI)

  SI units were used throughout the thesis.

- Communication (hardware, protocols, and algorithms):

  Communication/configuration (OSI (Open Systems Interconnection model): Transport Layer (ISO/IEC 7498)

  The simulations and technical discussions in this independent work refer to TCP (Transmission Control Protocol) and AQM (Active Queue Management), which operate in the OSI model Transport Layer.

- Programming languages & frameworks:

  C++ (SO/IEC 14882)

  Python

  Docker

  Pytorch

  The simulator code was developed in C++ using the Network Simulator 3 library, which is natively written in C++. Data was post-processed and plotted using Python code. The Pytorch library was used to develop the RL Agent. All code was run in a Docker image to overcome dependency issues between the NS–3 library `waf` build framework and the Pytorch library.

B Code

Much of the code from this work was adapted from the ABM code and RL-ABM code developed by Vamsi Addanki and Haiyue Ma & Hengrui Zhang, respectively. The Github repository for this thesis can be found here: `https://github.com/artemisveizi/ns3-rlbm-sb-compare.git`

C Supplemental Figures

Queue length plots (like those in Figures 13–15) can be found in `figure_rlbuffer/`, here: `https://github.com/artemisveizi/ns3-rlbm-sb-compare.git`

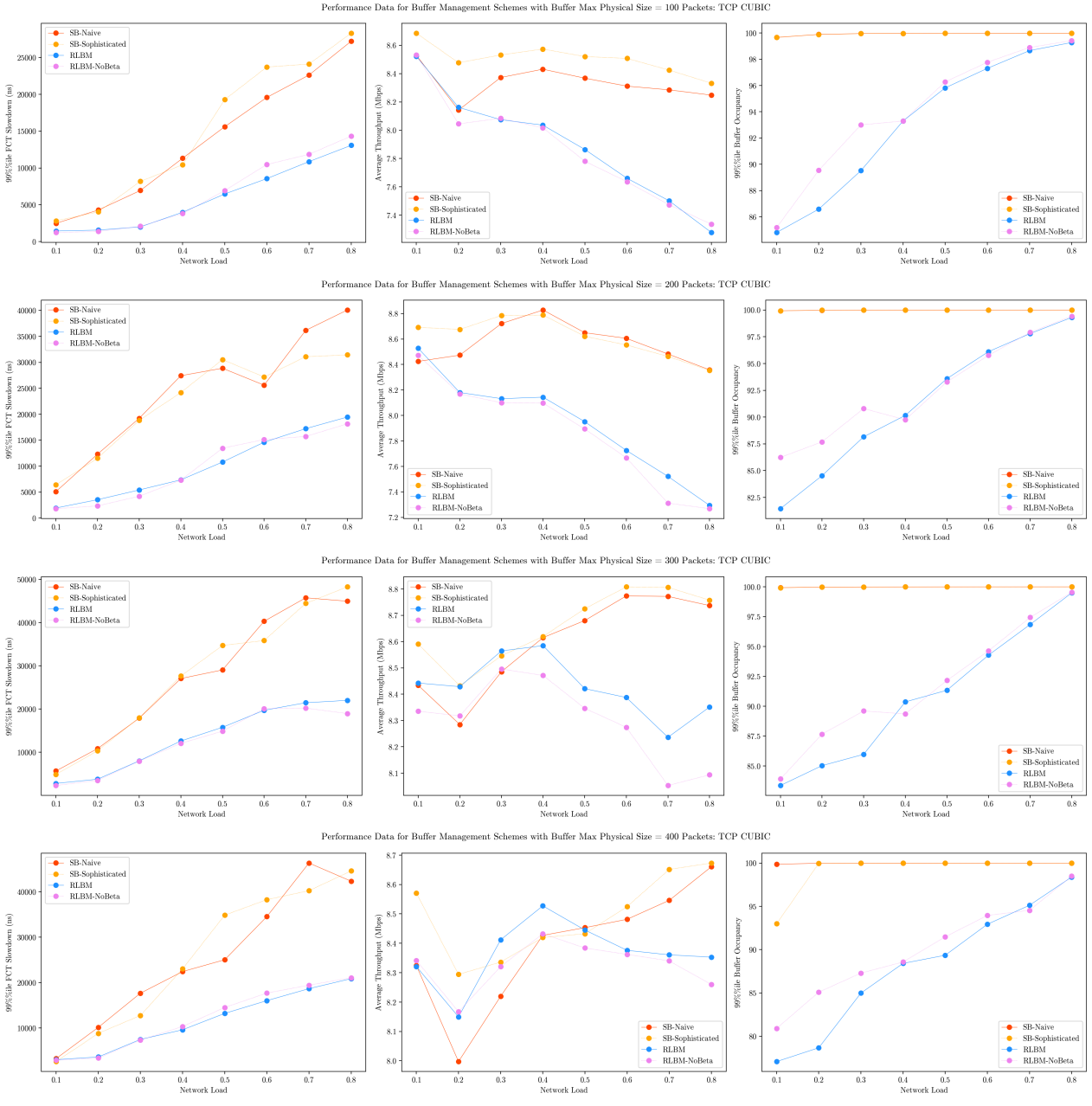

Figure 17: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP CUBIC Senders and Varying Buffer Physical Max Sizes (100 - 400 Packets).

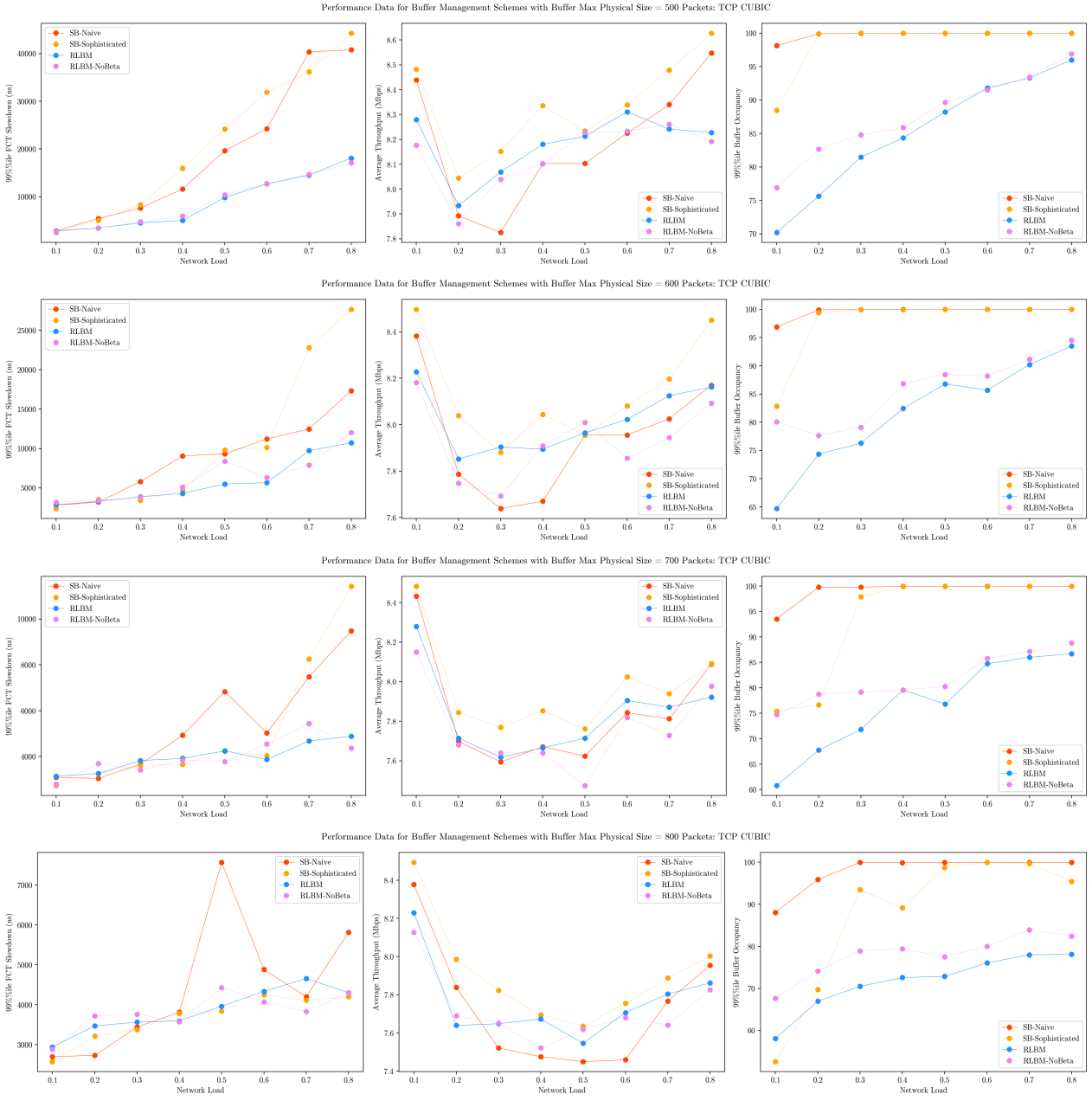

Figure 18: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP CUBIC Senders and Varying Buffer Physical Max Sizes (500 - 800 Packets).

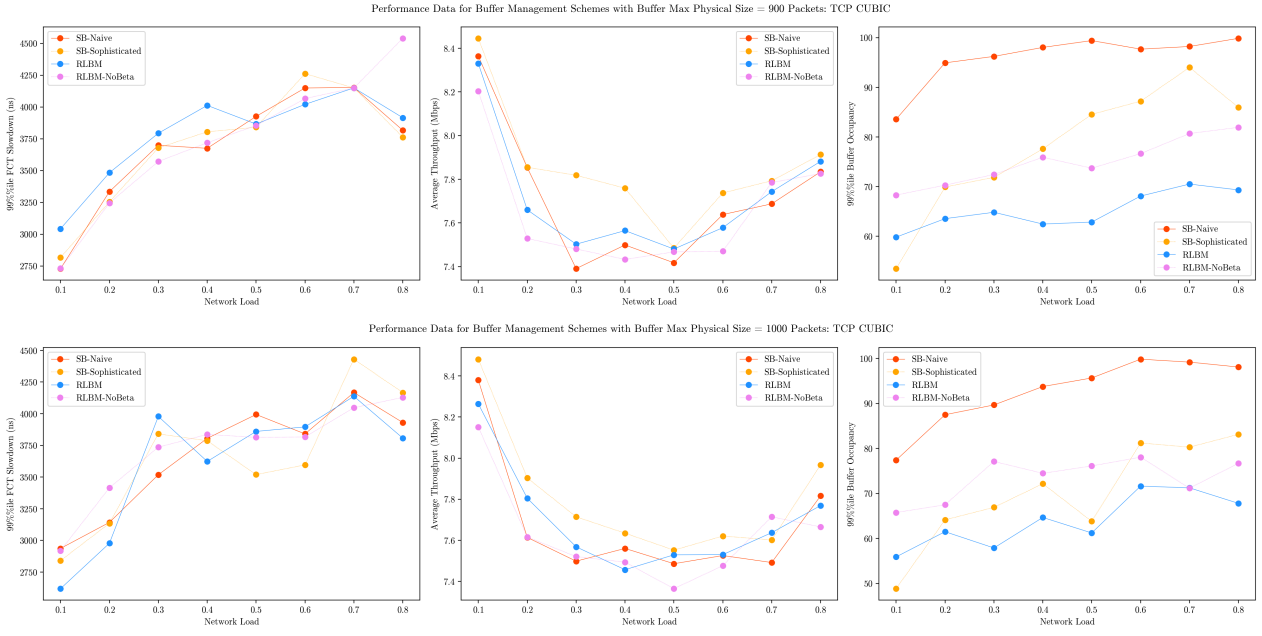

Figure 19: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP CUBIC Senders and Varying Buffer Physical Max Sizes (900, 1000 Packets).

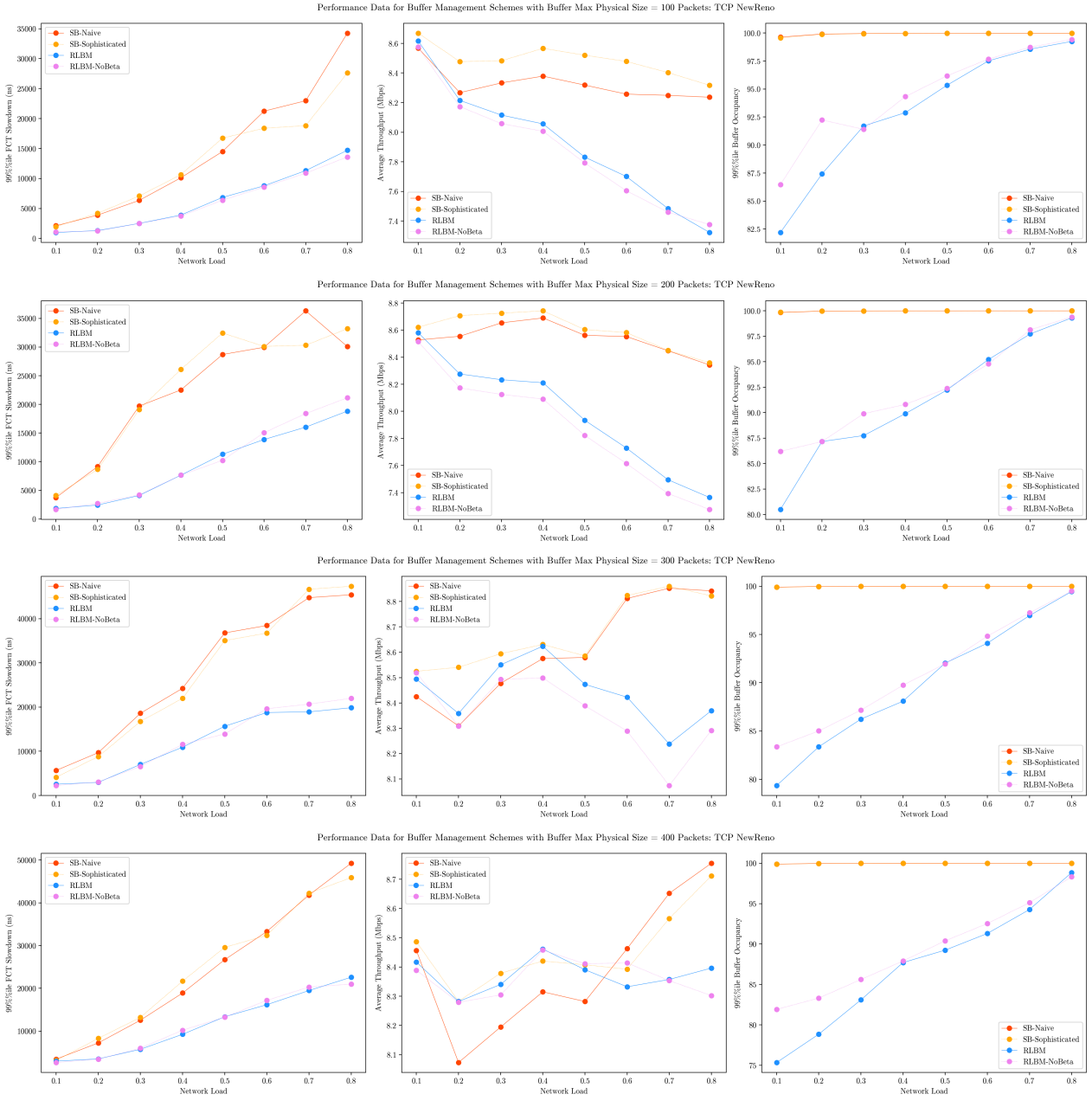

Figure 20: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP NewReno Senders and Varying Buffer Physical Max Sizes (100 - 400 Packets).

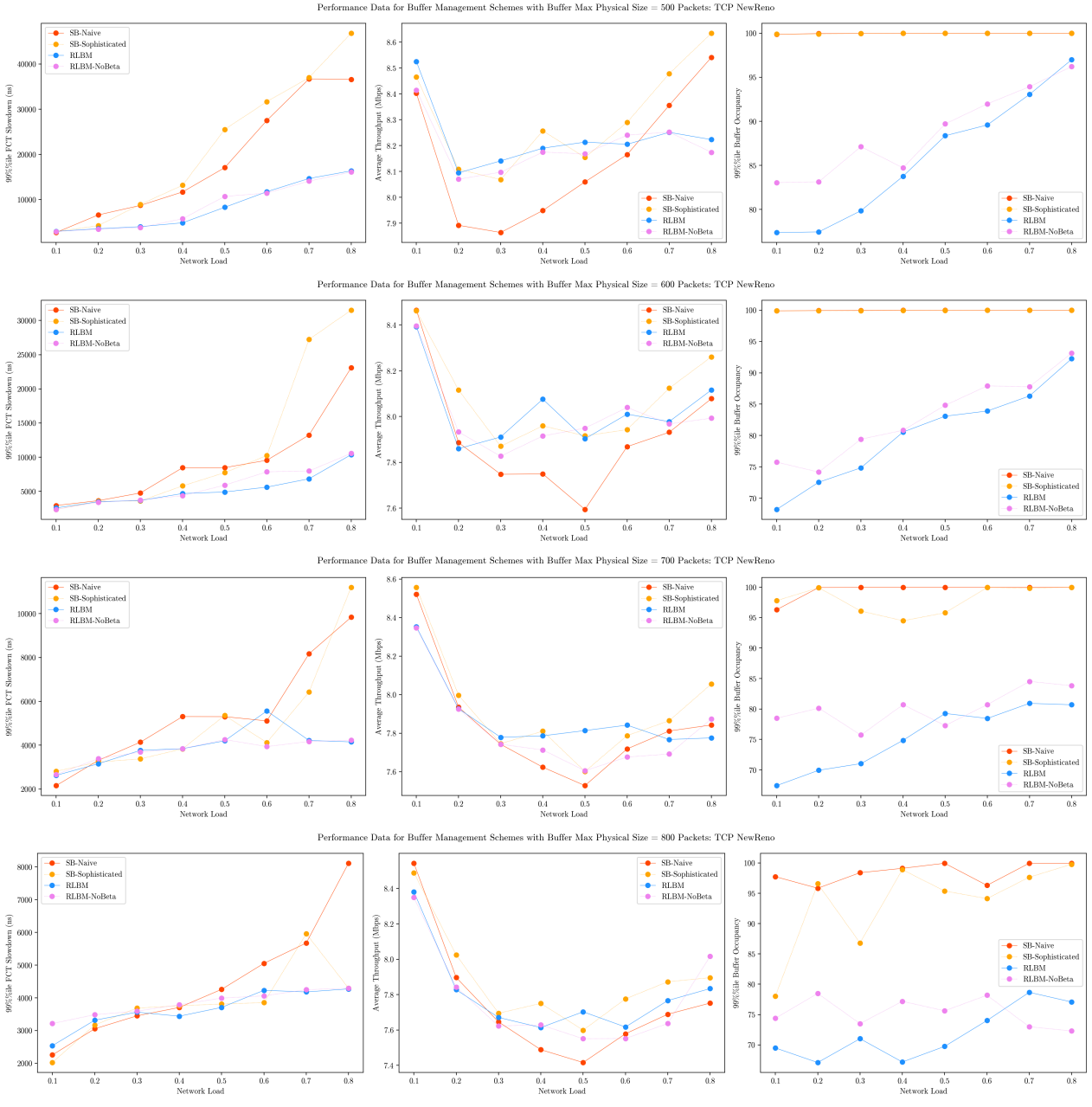

Figure 21: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP NewReno Senders and Varying Buffer Physical Max Sizes (500 - 800 Packets).

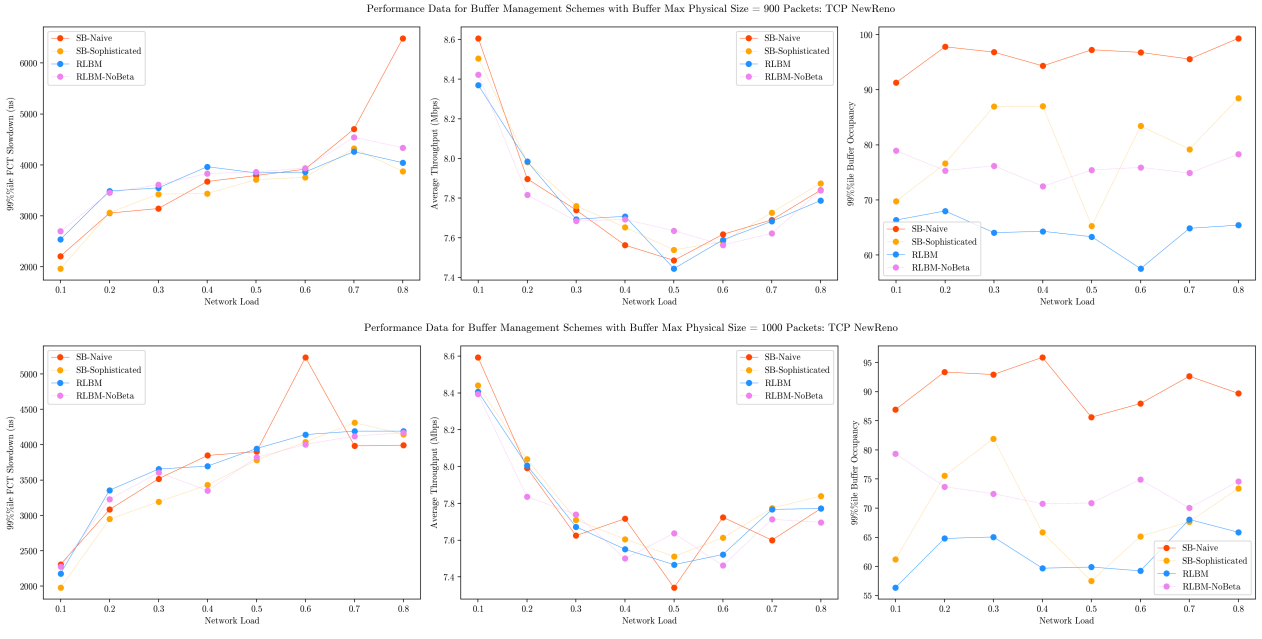

Figure 22: 99th Percentile FCT Slowdown (ns), Average Throughput (Mbps), and 99th Percentile Buffer Utilization (%) Results for Static Buffer: Naive, Static Buffer: Sophisticated, RLBM, and RLBM: No β for TCP NewReno Senders and Varying Buffer Physical Max Sizes (900, 1000 Packets).

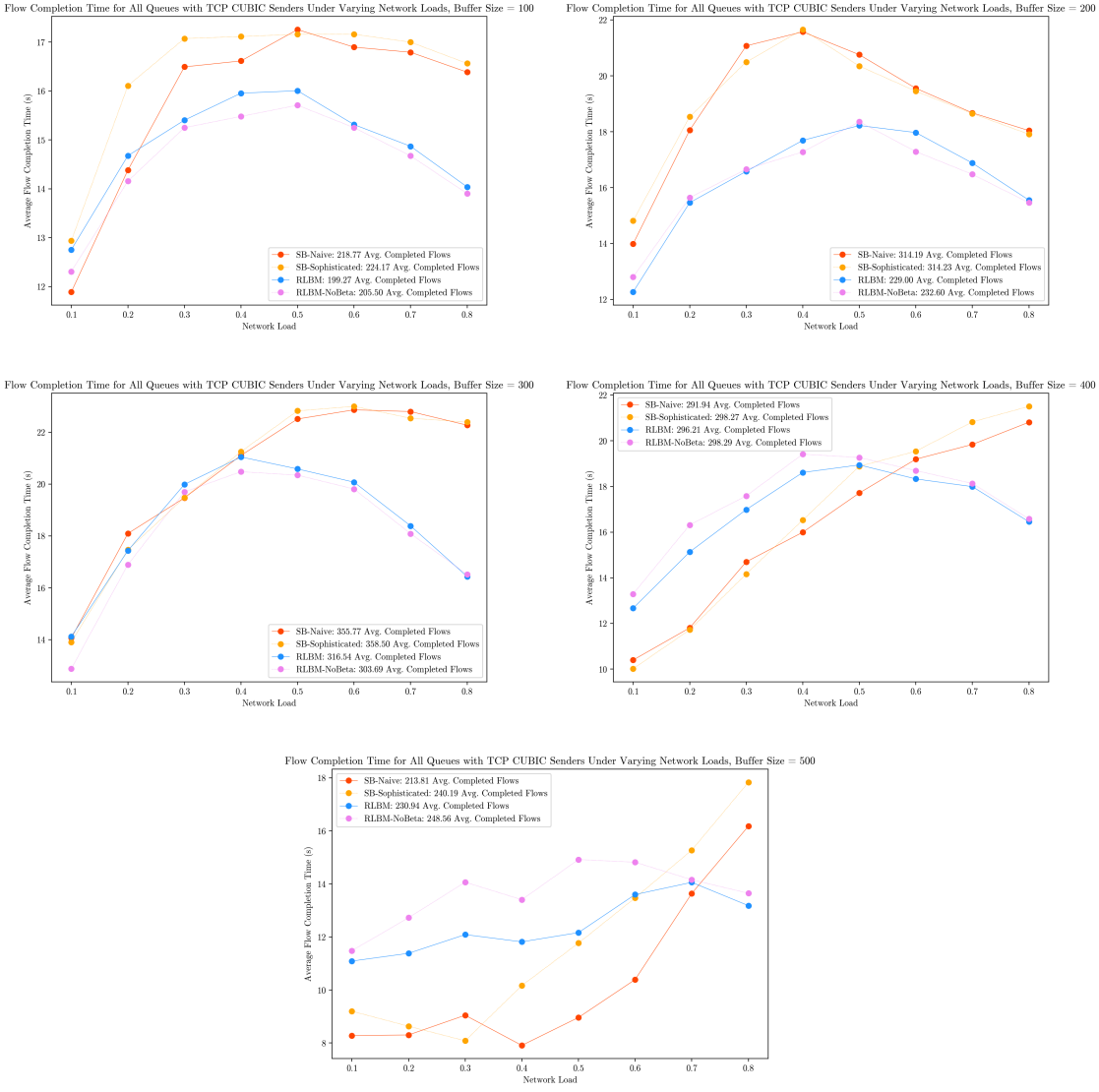

Figure 23: Average flow completion time of each BM strategy under varying Network Loads for Senders configured with TCP CUBIC; Buffer Sizes in 100–500 packets.

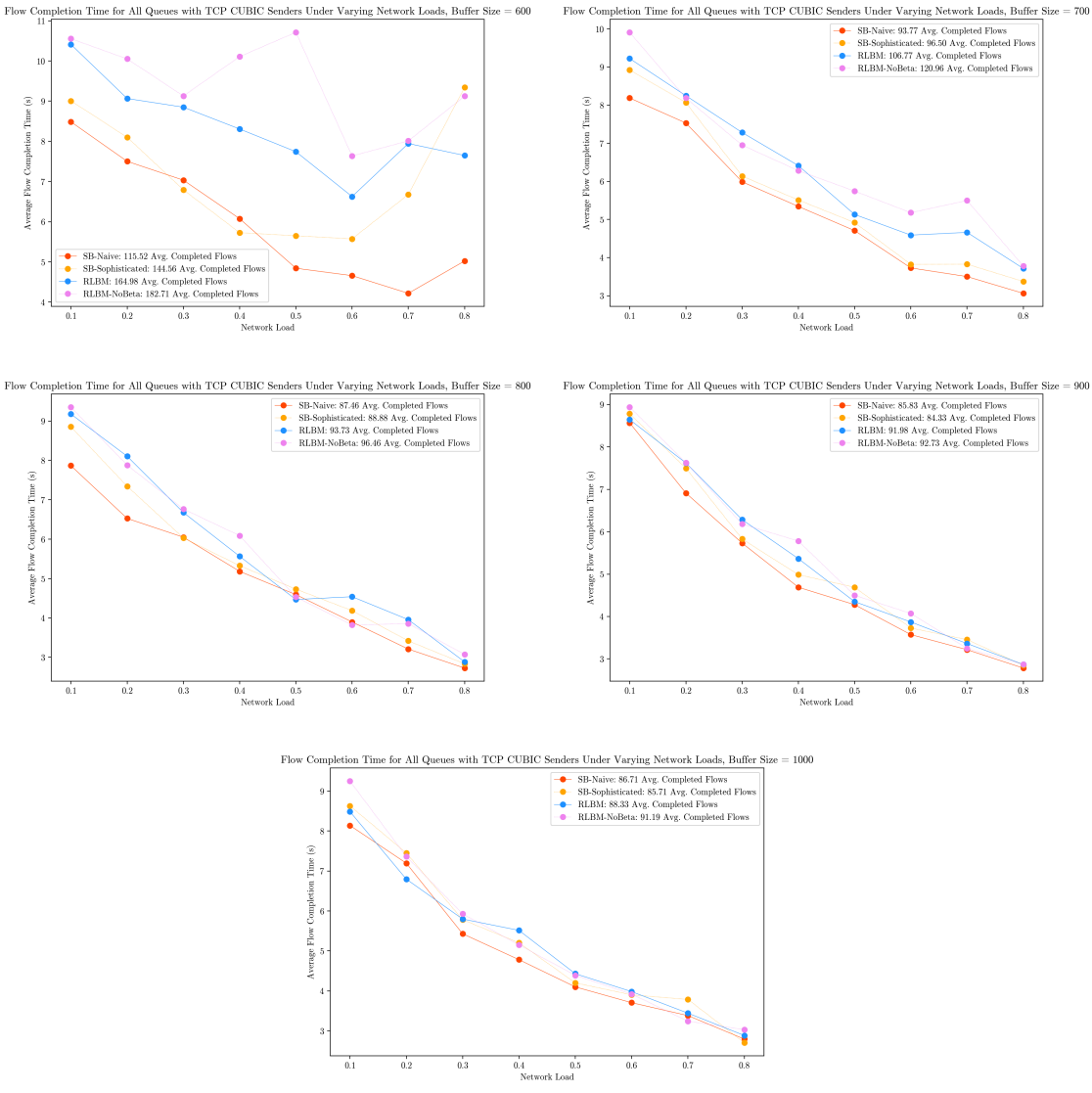

Figure 24: Average flow completion time of each BM strategy under varying Network Loads for Senders configured with TCP CUBIC; Buffer Sizes in 600–1000 packets.

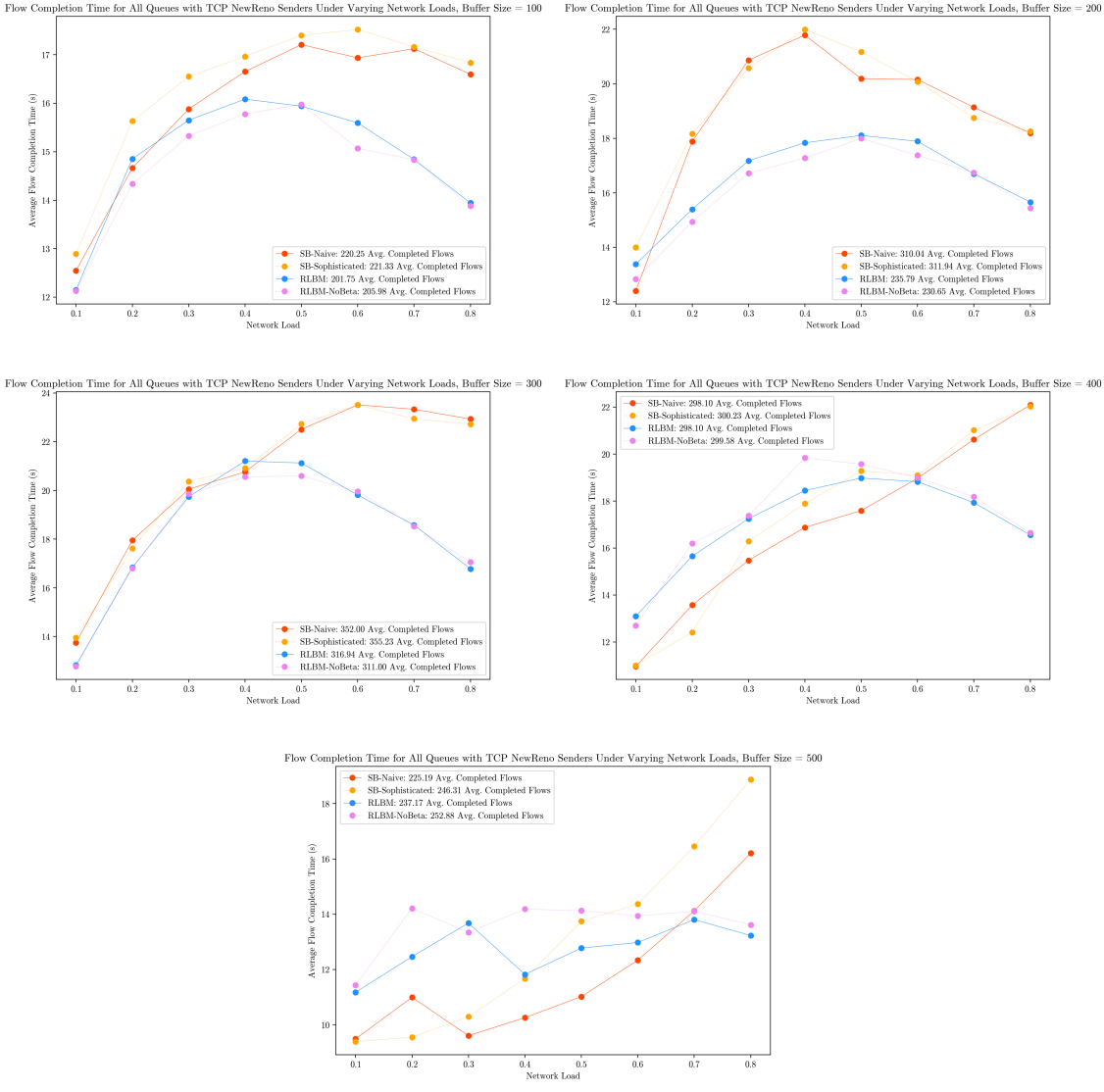

Figure 25: Average flow completion time of each BM strategy under varying Network Loads for Senders configured with TCP NewReno; Buffer Sizes in 100–500 packets.

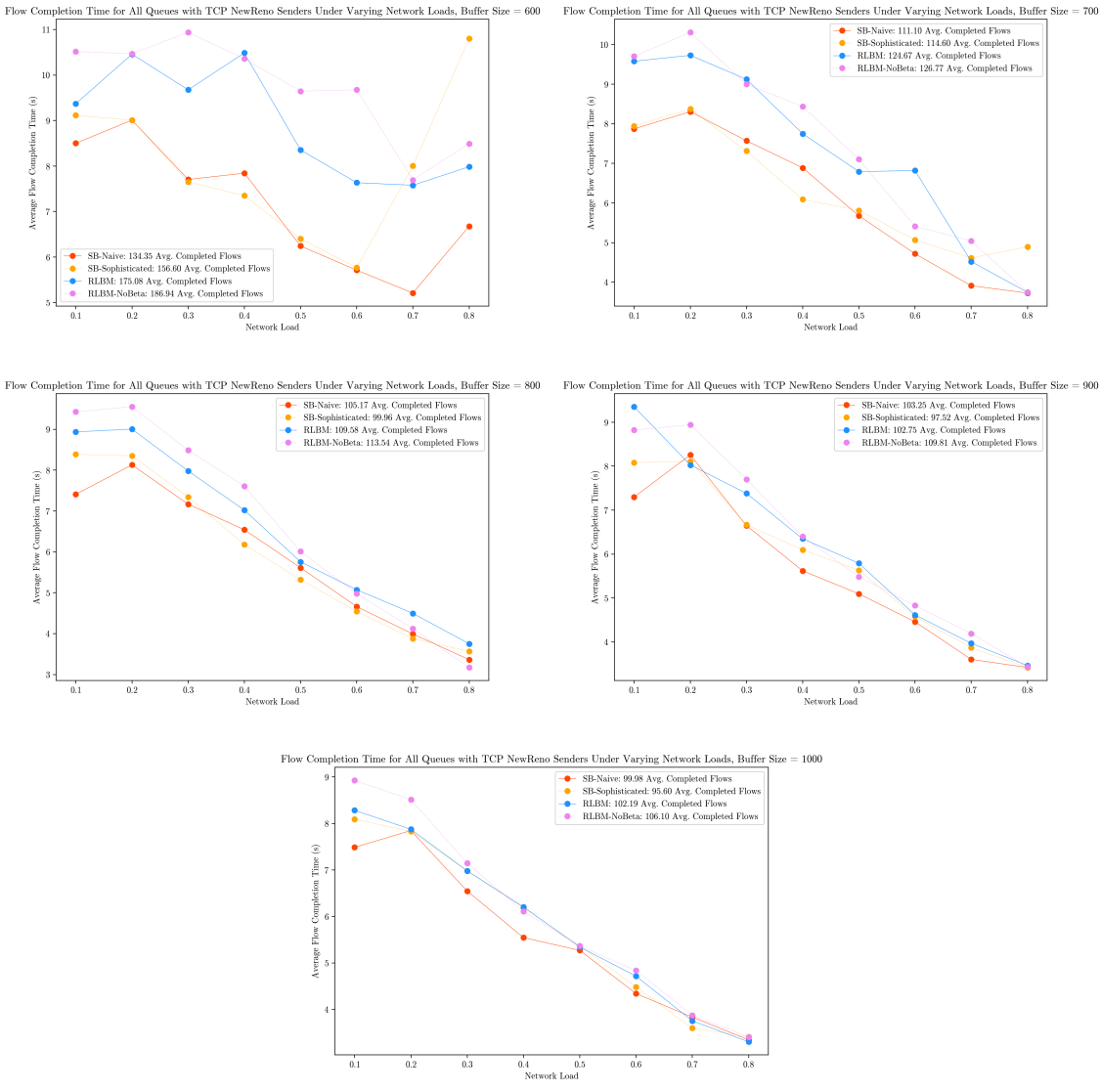

Figure 26: Average flow completion time of each BM strategy under varying Network Loads for Senders configured with TCP NewReno; Buffer Sizes in 600–1000 packets.