RV Educational Institutions ®
RV College of Engineering ®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi, Accredited By NAAC, Bengaluru And NBA, New Delhi

Go, change the world

Buffer optimization in data plane for Software Defined Networks using Reinforcement Learning Technique

MINOR PROJECT (18CS64)

Submitted by

DINESH BABU S 1RV18CS053

FURQAN ABDUL KHADAR RAMADURG 1RV18CS054

SRIRAM NARAYANA CUMMARAGUNTA 1RV18CS173

Under the guidance of

Prof. Sneha M,
Assistant Professor.

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

Academic Year 2020-21

RV College of Engineering

Bengaluru 560 059

Autonomous Institution Affiliated to VTU, Belagavi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the minor project work titled 'Buffer optimization in data plane for Software Defined Networks using Reinforcement Learning Technique' is carried out by Dinesh Babu S(1RV18CS053), Furqan Abdul Khadar Ramadurg(1RV18CS054), Sriram Narayana Cummaragunta(1RV18CS173) in partial fulfilment of the completion of the 6th Semester Minor Project18CS64 who are bonafide students of RV College of Engineering®, Bengaluru, in partial fulfilment for the award of degree of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belagavi during the academic year **2020-21**. It is certified that all corrections/suggestions indicated for the internal Assessment have been incorporated in the report deposited in the departmental library. The Minor Project report has been approved as it satisfies the academic requirements in all respect as prescribed by the institution for the said degree.

Signature of Guide	**Signature of HoD**	**Signature of Principal**
Prof. Sneha M	Dr.Ramakanth Kumar P	Dr. Subramanya. K. N

External Examination

Name of Examiners	**Signature with date**
1.	
2.	

Acknowledgement

Any achievement, be it scholastic or otherwise does not depend solely on the individual efforts but on the guidance, encouragement and cooperation of intellectuals, elders and friends. A number of personalities, in their own capacities have helped me in carrying out this project work and, we would like to take this opportunity to thank them all.

We deeply express our sincere gratitude to my guide **Prof. Sneha M**, Assistant Professor, Department of Computer Science and Engineering, RVCE, Bengaluru, for her able guidance, regular source of encouragement and assistance throughout this project.

We would like to thank **Dr.Ramakanth Kumar P**, Head of Department, Computer Science and Engineering, R.V.C.E, Bengaluru, for his valuable suggestions and expert advice.

First and foremost we would like to thank **Dr. Subramanya. K. N**, Principal, R.V.C.E, Bengaluru, for his moral support towards completing our project work.

We thank our Parents, and all the Faculty members of Department of Computer Science and Engineering for their constant support and encouragement.

Last, but not the least, we would like to thank our peers and friends who provided us with valuable suggestions to improve our project.

Abstract

Software-defined networking (SDN) technology is an approach to network management that enables dynamic, programmatically efficient network configuration in order to improve network performance and monitoring, making it more like cloud computing than traditional network management. SDN is meant to address the fact that the static architecture of traditional networks is decentralized and complex while current networks require more flexibility and easy troubleshooting. SDN attempts to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane). The control plane consists of one or more controllers, which are considered the brain of the SDN network where the whole intelligence is incorporated. However, the intelligent centralization has its own drawbacks when it comes to security, scalability and elasticity and this is the main issue of SDN.

SDN structure is fairly similar to an Operating System. The advantage here compared to traditional switches is that there is a centralization of commands. The basic idea of this project is to automate the buffer optimisation process depending on the traffic and also to reduce latency.

The technology that can be used for the implementation here is Reinforcement learning. With the help of this, the agent (i.e. controller (here)) can be trained to take decisions based on the type of traffic in the network.

Whilst there is a direct correlation between the throughput and the delay, the RL agent is reacting to the environment and accordingly throttling the queue size of the switch. This in turn leads to increased throughput and decreased delay/latency. The results obtained show a stark change in the performance metrics with the corresponding change in environment.

With this project, we conclude that the usage of a Reinforcement Learning agent to vary the buffer size of a switch in the data plane of an SDN would cause significant improvements in delay and throughput. This project currently simulated UDP communication, as the concept of acknowledgement packets in this simulation would have led to unnecessary complications. An improvement would be to switch to TCP.

Contents

List of Figures

1 Introduction

Software Defined Networking (SDN) design is fairly similar to that of an Operating System (OS). Like the Operating System even this has a controller that interacts with hardware. As illustrated in Figure 1, the hardware here are forwarding devices. The controller communicates with the network application too (like application software in OS).

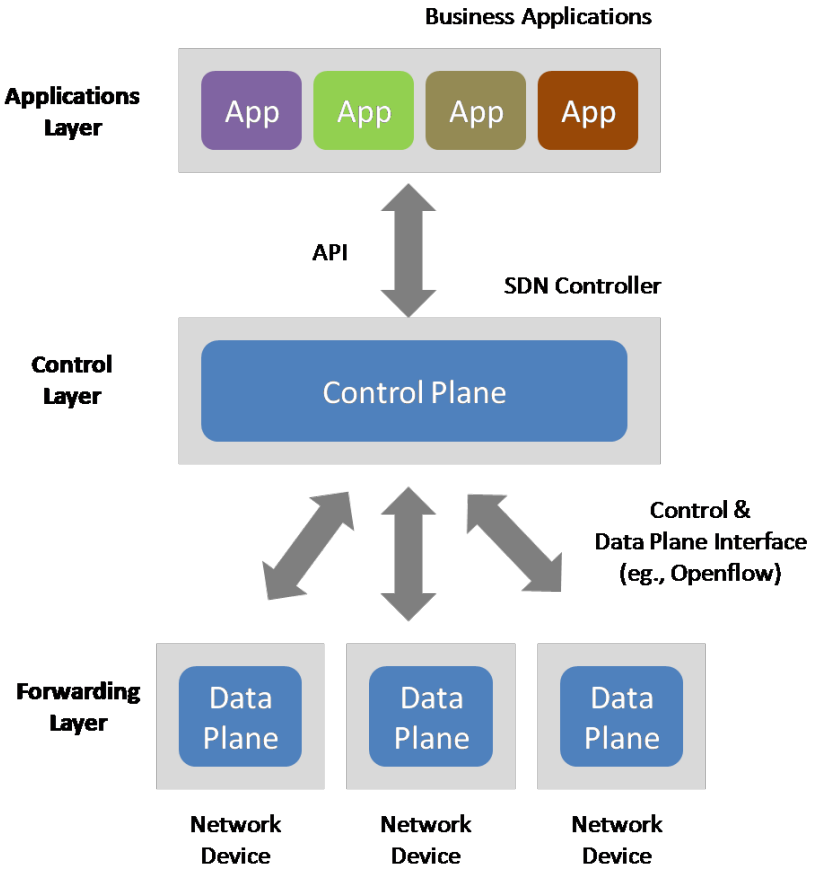


Figure 1: SDN Architecture

There exists different protocols for communication between the controller and forwarding device and also between controller and Network applications. The OpenFlow protocol is commonly used in communication between forwarding device and controller. In contrast to the traditional switch which has only two planes Data and Control. The data plane can't be accessed directly and only the interface in the control plane allows it to make certain calls or use some functions to access or modify the data plane. It is also cumbersome to set up a network of traditional switches compared to setting up an SDN.

1.1 State of Art Developments

Software defined networking (SDN), one of the predominant and relatively new networking paradigms, seeks to simplify network management by decoupling network control logic from the underlying hardware and introduces real-time network programmability enabling innovation.

The present work reviews the state of the art in software defined networking providing a historical perspective on complementary technologies in network programmability and the inherent shortcomings which paved the way for SDN.

1.2 Motivation

The motivation for this project developed from noticing the problem of finding the optimal size of buffer to be used for the traffic because the traffic keeps changing from time to time so the optimal buffer length must also change accordingly. This is a very frustrating job if asked to do manually. The goal of this project is to design an Agent with the use of Reinforcement Learning. The agent will be the controller which will learn from the traffic and the decisions taken in those situations. The end result of this project will help solve the problem of buffer optimisation with less latency.

1.3 Problem Statement

Develop a Reinforcement Learning model to optimize the buffer sizes of switches in the data plane of an SDN.

1.4 Objectives

The objective of this project is to:

1. Reduce the latency in Software Designed Networks (SDNs) while finding a balance with throughput
2. Achieve this by buffer size alteration in the data plane using Reinforcement Learning

1.5 Methodology

Initially, the team tried to simulate SDNs on Mininet using POX controller and Ryu controllers. After tasting limited success, the team then decided to focus on developing an RL Agent that would be ready to integrate with an SDN once found. On completion on development of the Reinforcement Learning Agent that would facilitate buffersizing, the team then continued to work on Python language Scripts that simulated a network using Discrete Event Simulation. The Discrete Event Simulation strategy led the team to the realisation of the project at the time of writing this report.

1.5.1 Discrete Event Simulation - Working Strategy

1. Explore static buffersizing by plotting the performance metrics against different switch buffer sizes

2. Enable dynamic buffersizing by interfacing the Discrete Event Simulation with the Reinforcement Learning Agent

3. The Reinforcement Learning model ensures that the agent is reward for increase in throughput and decrease in delay and vice versa

4. The data points are then stored and plotted to analyse the effects of dynamic buffersizing on packet drop, delay and throughput

1.6 Summary

The research done on the state of the art development in SDN and reinforcement learning and the methodology helped the team develop a suitable software that will encompass the requirements of the project. In the process, the team gained knowledge on Ryu and POX controllers, usage of MATLAB and Simulink for simulation of network topologies and much more.

2 Literature Survey

2.1 Introduction

Over 20 research papers were consulted as part of the knowledge survey for this project. Of those many of them had similar findings and limitations, and many did not focus on the specific topic of Buffer optimisation in SDN. However, the myriad papers helped in gaining some context and knowledge about SDNs, Reinforcement Learning and other project relevant topics.

2.2 Related Work

The paper on Buffer Management of Virtualized Network Slices for Quality-of-Service Satisfaction by Kim et. al., [5] says that using SDN and Network Functions Virtualization (NFV), network slicing creates programmable and flexible network instances on a shared network infrastructure. Network slicing also maximizes flexibility and fulfilling the diverse needs of services. The limitations of the paper was that it does not address latency and throughput in detail. No machine learning techniques are used as the paper takes a purely mathematical approach.

The paper on Applying Buffer to SDN Switches by Li et. al., [6] says that using the intrinsic buffer in a SDN switch can also greatly reduce the communication overhead without using additional devices. If a switch buffers each mismatch packet, only a few header fields instead of the entire packet are required to be sent to the controller. The limitation was that QoS guarantees have not been demonstrated for diverse applications

The paper on Pausing and Resuming Network Flows using Programmable Buffers by Lin et. al.,[7] talks about how existing Software Defined Network (SDN) solutions fail to provide sufficient decoupling between the data-plane and control-plane to enable efficient control of where, when, and how a network flow is buffered without causing excessive control-plane traffic. The limitation was that Programmable Buffers might not always be effective due to its reliance on e.g. low control latency (controller proximity).

The paper on Practical and Dynamic Buffer Sizing using LearnQueue by Bouacida et. al.,[2] was incredibly promising in that it provided valuable insight into Reinforcement Learning on queues and practically provided a roadmap for the project. It speaks about how the LearnQueue design periodically tunes the queue size by predicting the reward or the penalty resulting from explored actions along with exploiting the prior experience accumulated. However, more sophisticated exploration/exploitation strategies in correlation with the dynamics of the queue need to be employed.

Mondal et. al., [8] address the problem of minimum buffer size evaluation of an OpenFlow system in SDNs, while ensuring optimum packet waiting time in their paper "Buffer Size Evaluation of OpenFlow Systems in Software-Defined Networks". However, the Queuing model needs to be improved and TCAM memory needs to be properly utilised.

It is illustrated by Latah et. al., in "Applying Buffer to SDN Switches" [6] that the research community has showed an increased tendency to benefit from the recent advancements in the artificial intelligence (AI) field to provide learning abilities and better decision making in SDN. In this study, they provide a detailed overview of the recent efforts to include AI in SDN. However, more efforts towards studying the robustness of AI approaches under adversarial settings need to be take into consideration, as well.

Kaljic et. al., have illustrated in A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking [3] that by establishing a correlation between the treated problem and the problem-solving approaches, the limitations of ForCES and OpenFlow data plane architectures were identified. Based on identified limitations, a generalization of approaches to addressing the problem of data plane flexibility and programmability is made. However, simple extensions cannot solve problems of programmability and flexibility of the existing data plane architectures

Existing queuing models for SDN have focused on the switches that immediately send packets to the controller for decisioning. In Modelling Switches with Internal Buffering in Software-Defined Networks, Singh et. al., [9] propose an analytical model for SDN switch with the internal buffer to investigate the potential of internal buffering in SDN switches. Costs of memory needs to be balanced with the costs of meeting QoS requirements based on the analytical predictions given by the models in this paper.

For efficient network management, Active Queue Management (AQM) has been proposed by Minsu Kim [4] which is the intelligent queuing discipline. In this paper, we propose a new AQM based on Deep Reinforcement Learning (DRL) to handle the latency as well as the trade-off between queuing delay and throughput. We choose Deep Q-Network (DQN) as a baseline of our scheme, and compare our approach with various AQM schemes by deploying them on the interface of fog/edge node in IoT infrastructure. The paper does not tackle buffer management specifically. It tackles Queueing and Dequeuing for IoT networks. The simulation needs a more complicated topology with multiple switches. Not a single one.

Fine tuning of the buffer size is well known technique to improve the latency and throughput in the network. However, it is difficult to achieve because the microscopic traffic pattern changes dynamically and affected by many factors in the network, and fine grained information to predict the optimum buffer size for the upcoming moment is difficult to calculate. To address this problem, Sneha et. al.,[10] propose a new approach, Buffer Optimization using Reinforcement Learning (BO-RL), which can dynamically adjust the buffer size of routers based on the observations on the network environment including routers and end devices. This paper was immensely helpful in gaining insight into the concept of SDNs, buffersizing and reinforcement learning.

2.3 Summary

Extensive literature review has been conducted, encompassing relevant topics such as Deep Queue Learning and QoS management for various SDN configurations from reputable journals. These reviews have helped attempt to implement this idea in practice. In addition to the publications mentioned here, various sources on the different simulation platforms and SDN controllers were explored and analysed. All the exploration has led to the team reaching inform decisions with respect to the roadmap for the project at hand.

3 Software Requirements Specification

The Software Requirement Specification document also called the SRS, helps organise all the requirements that need to be met once the software is developed. The functional requirements describe the operations that various classes of users should be able to perform, be it general access or special access control oriented. The non-functional requirements are those that impose performance constraints that the system should be able to meet to ensure smooth operation.

3.1 Functional Requirements

The functional requirements can be classified based on the user classes as follows: Please note that there is only one User class and the functional requirements for the same are given below:

1. Create topology specified
2. Populate network
3. Change the Queue size
4. Run Reinforcement Learning (Q-Learning) on the buffer
5. Notice change in buffer size
6. Give a tangible measure for the efficiency of the RL Algorithm (Latency and Throughput)

3.2 Non-Functional Requirements

1. **Scalability**: The system is highly scalable.
2. **Availability**: The system is to be available for anyone who is eligible/verified to perform the necessitated tasks.
3. **Reliability**: The system must be able to accurately perform the task of buffer optimisation
4. **Recoverability**: The system is centralized which makes it easy to coordinate with forwarding devices and also error recovery is easy.
5. **Integrity**: The data that is stored cannot be overwritten, and hence the data integrity is maintained.

3.3 Hardware and Software requirements

3.3.1 Hardware Requirements

- RAM: 4 GB and above
- Any Intel or AMD x86-64 processor
- 5-8 GB of free disk space

3.3.2 Software Requirements

- Windows 7+ or Linux or MacOS
- Python 3.6 or above

3.3.3 Software Dependencies

- matplotlib
- numpy
- simpy
- heapq
- functools

3.4 Summary

It can be noticed that all the requirements mentioned above, functional as well as non-functional can be facilitated through use of technology as it natively upholds the principles that surround these requirements. More requirements can be incorporated if need be during the course of development. More about this has been illustrated in the Future Scope section of this report. The user would be happy to note that this project uses the bare minimum in terms of hardware and software requirements and can be run easily on most available present day systems.

4 Design of Buffer optimization in data plane for Software Defined Networks using Reinforcement Learning Technique

4.1 High-Level Design

The high level design illustrates the various components of the system and how they interact with each other. It provides a general outline of the workflow that is expected to happen in the developed application.

4.1.1 System Architecture

Network architecture is represented in Figure 2. The elements of the network topology as as follows:

- Packet: This is a simple data object that represents a packet. Generally created by PacketGenerators. Key fields include: generation time, size, flow_id, packet id, source, and destination. We do not model upper layer protocols, i.e., packets don't contain a payload. The size (in bytes) field is used to determine transmission time.
- PacketGenerator: A PacketGenerator simulates the sending of packets with a specified inter-arrival time distribution and a packet size distribution. One can set an initial delay and a finish time for packet generation. In addition one can set the source id and flow ids for the packets generated. The packet generators out member variable is used to connect the generator to any component with a put() member function.
- PacketSink: Packet sinks record arrival time information from packets. This can take the form of raw arrival times or inter-arrival times. In addition the packet sink can record packet waiting times. Supports the put() operation.
- SwitchPort: Models a first-in, first-out (FIFO) queued output port on a packet switch/router. You can set the rate of the output port and a queue size limit (in bytes). Keeps track of packets received and packets dropped.
- PortMonitor: A PortMonitor is used if you want to monitor queue size over time for a SwitchPort. You need to specify a sampling distribution, i.e., a distribution that gives the time between successive samples.

The interpretation of the figure is as follows: PG stands for Packet Generator and as the name suggests, it represents the origin for the UDP packets in the network. There are three such Packet Generators in the network connected to switches S1, S4 and S3 respectively. There are four switches and to Packet Sinks. Packet Sinks are denoted by PS, and denote an endpoint for the UDP packets. The concept of generators and sinks exists in order to determine the various performance metrics for the network.

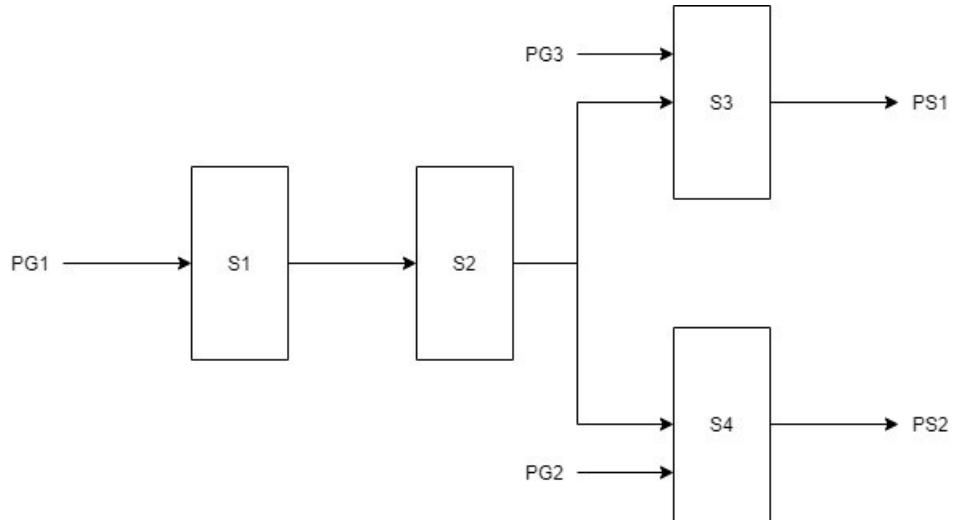


Figure 2: Network Architecture

4.1.2 Dataflow Diagrams

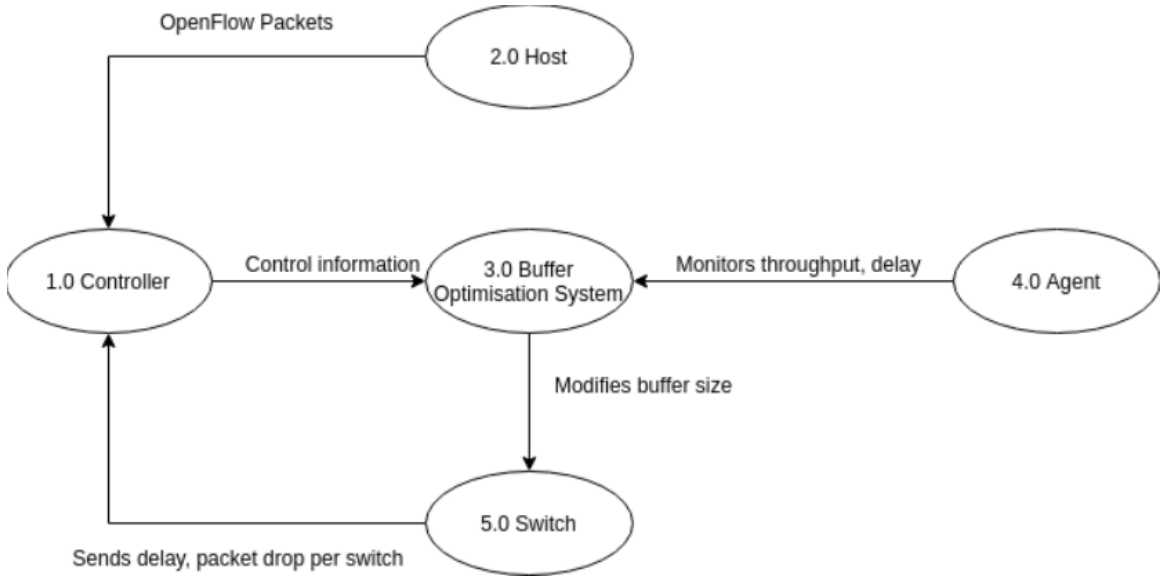


Figure 3: Dataflow Diagram - Level 0

Figure 3 illustrates the Level 0 Dataflow Diagram for the project. The controller collects control information and passes it on to the Buffer Optimization System. The Reinforcement Learning Agent monitors throughput and delay and passes the information on to the Buffer Optimization System as well. The system modifies the buffer of the switch which in turn sends delay and packet drop characteristics to the controller. All this information is collected from a network where a host sends the packets to a switch which then sends it to another host through other switches.

Figure 4 illustrates the Level 1 Dataflow Diagram for the project. Here, the roles of the Cognitive Agent is clearly demarcated. Besides, the diagram communicates about how there is data flow between a host and another host through a switch and how the controller orchestrates

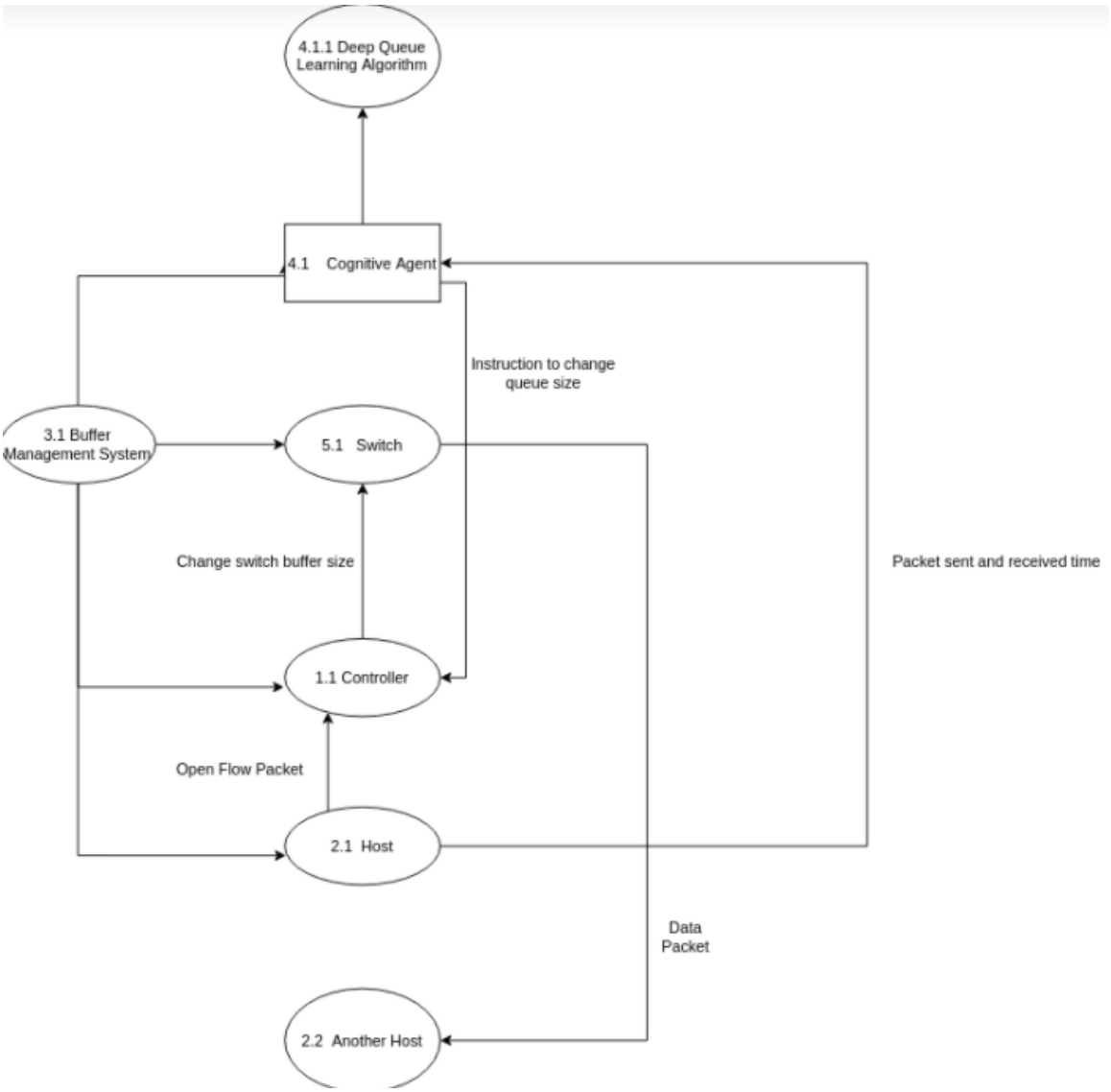


Figure 4: Dataflow Diagram - Level 1

this process. We also are able to witness that in order to make calculations for the performance metrics, the packet sent and received time is also being taken care of.

4.2 Detailed Design

Here, we see a simple structure for the working of the Reinforcement Learning model. In this Active Queue Management model in Figure 5, the state of the environment coupled with the reward calculated for the state act as input to the Agent. The agent then ascertains an action for it to undertake on the Queue. As in, should it dequeue a packet, etc.

4.2.1 Structure Chart

Figure 6 simplistically describes the overall structure of the project. Once the initial buffer size is decided using manual testing, the iterative process trains the Reinforcement Learning Agent

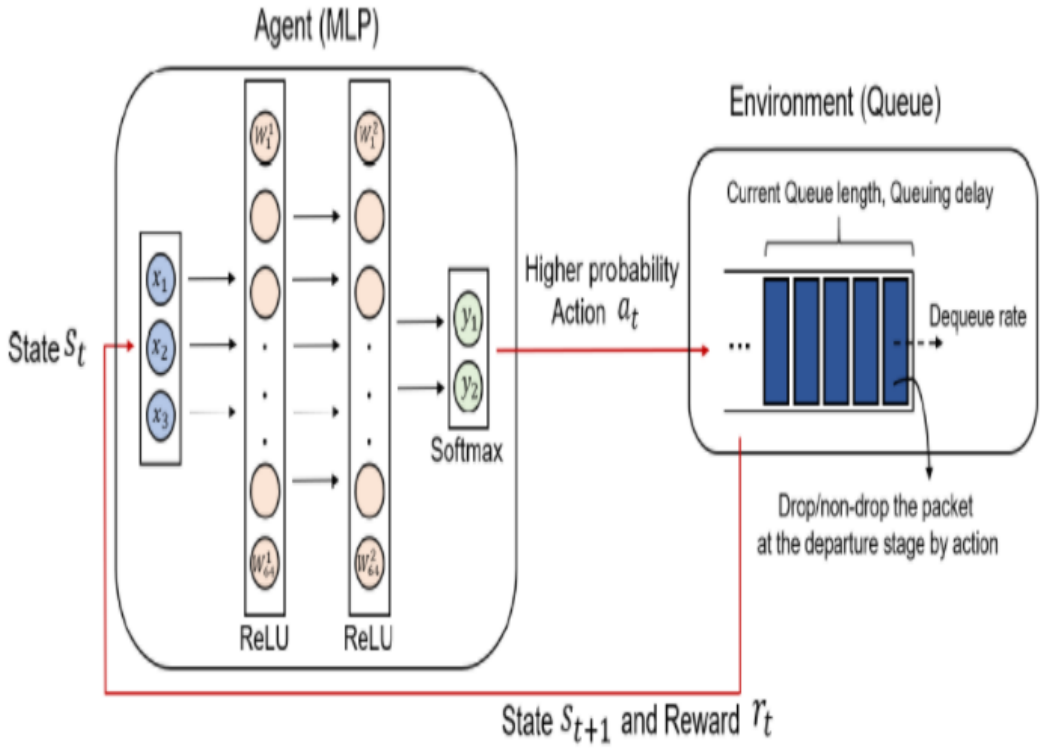


Figure 5: Representation of QLearning

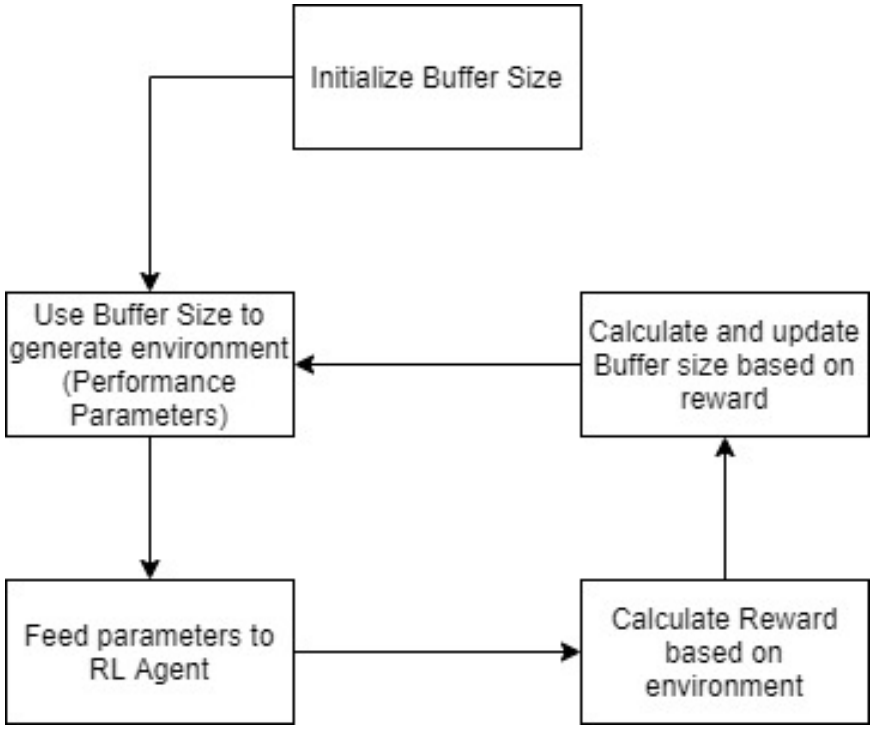


Figure 6: Structure Chart

to make decisions that would help optimize the performance metrics. This ends up looking like a cyclical process.

4.2.2 Functional Description of the Modules

The Discrete Event Simulation code creates a structure for the different element in the network. This structure is then used to simulate and calculate performance metrics. When dynamic buffersizing is enabled, the Reinforcement Learning model kicks in and calculates the rewards and gives it back to the DES code. The code then uses this reward parameter to calculate the new buffer size in bytes.

The Discrete Event Simulation code consists of three separate modules. The first one is called SimComponents and it serves as a template for the creation of classes for different components in the network Simulation. The second part is called QueueNet2 and it helps calculate the different performance metrics using the parameters initialized in the SimComponents module. The third and final part of the puzzle is the driver module. This serves as the heart of the project and is the place of integration of the RL model in the DES Code. It is the module where the dynamic buffersizing occurs.

4.3 Summary

The complete architecture of the system has been depicted in the form of different architectural designs. The High-Level Design gave a glimpse into the network architecture and introduced the reader to the various components of a network. It also illustrated the flow of data in the project through Dataflow Diagrams. The detailed design has illustrated how an RL Agent works and the structure chart helps give a functional overview of the process. Finally, the functional description of the respective modules gave insight into the working of each part of the project.

5 Implementation of Buffer optimization in data plane for Software Defined Networks using Reinforcement Learning Technique

5.1 Programming Language Selection

After considering the Mininet platform and the POX controller based on Python and simultaneously considering MATLAB, it was finally agreed upon that the team would use the Discrete Event Simulation concept to implement the code on python itself. The RL Algorithm being used is the DQN. It is a reinforcement learning algorithm that combines Q-Learning with deep neural networks to let RL work for complex, high-dimensional environments, like video games, or robotics.

5.2 Platform Selection

In network simulations there can be a number of tasks such as packet generation, transmission, queueing, that appear to run at the same time. In modern computers, operating system mechanisms such as processes and threads providing this illusion. For simplicity of implementation, ease of use, and scalability such mechanisms are not used for discrete event simulation (DES), instead techniques more akin to coroutines are frequently employed.

Although Python's generators provide functionality very similar to coroutines there is still a fair amount of work needed to create a discrete event simulation (DES) system. This is where SimPy, a very nice, open source, DES package comes in. SimPy is a general purpose DES package, not networking specific.

This was after the careful consideration of Mininet with POX controller and the Ryu controller. The team was not able to obtain any meaningful data from these platforms in order to train and integrate RL Agents.

5.3 Code Conventions

Pep-8 coding style has been maintained throughout the project in order to maintain styling and keep the code clean. In order to automate the process of formatting and cleaning up code smells, the project makes use of black and mypy. These are tools that automatically perform static analysis on the code and improve its quality.

5.4 Summary

Discrete Event Simulation Code written on the basis of Python's SimPy library has been used as a basis for this project [1]

6 Experimental Results and Testing

6.1 Evaluation Metrics

The evaluation metrics would be as delay, packet drop and throughput. Once these are plotted against the time taken for the process, the working of the project is visible, and conclusions can be drawn. Average delay is calculated by summing up the end to end delays of each packet and dividing it by the number of packets in the system. The packet drop probability is calculated using the number of packets sent and received. The switch packet drop is ascertained by a packet drop parameter assigned to each switch in the network. The throughput is calculated for every 4000 milliseconds. In fact, all data points in every graph in this section of the report is obtained every 4000 milliseconds.

6.2 Experimental Dataset

The experimental dataset is created using the following code in the program:

```
delay1, delay2, pdrop, sw_pdrop, throughput = QueueNet2.main(bsize)
pdrops.append(pdrop)
delay1s.append(delay1)
delay2s.append(delay2)
sw_pdrops.append(sw_pdrop)
throughputs.append(throughput)
```

Here, pdrop stands for Packet Drop, sw_pdrop stands for Switch Packet Drop. The variables delay and throughput are self-explanatory. Here are the structures for the Packet Generator and Sinks that are used to generate the data.

Packet Generator

```
self.id = id
self.env = env
self.adist = adist
self.sdist = sdist
self.initial_delay = initial_delay
self.finish = finish
self.out = None
self.packets_sent = 0
self.action = env.process(self.run())  # starts the run() method as a SimPy
                                        process
self.flow_id = flow_id
```

This code structure for Packet Generator has the following attributes: id and environment. There's an initial delay per packet because we can assume that the network being shown here is only a part of a whole. There's also the number of packets being sent, and the run() method being called to run the SimPy process.

Packet Sink

```
self.store = simpy.Store(env)
self.env = env
self.rec_waits = rec_waits
self.rec_arrivals = rec_arrivals
self.absolute_arrivals = absolute_arrivals
self.waits = []
self.arrivals = []
self.debug = debug
self.packets_rec = 0
self.bytes_rec = 0
self.selector = selector
self.last_arrival = 0.0
```

This code structure for Packet Generator has the following atrributes: It keeps track of the arrivals using rec_arrivals and absolute_arrivals. The waits array stores the total waiting times for the packets. packets_rec and bytes_rec determine the number of packets and bytes received respectively.

6.3 Performance Analysis

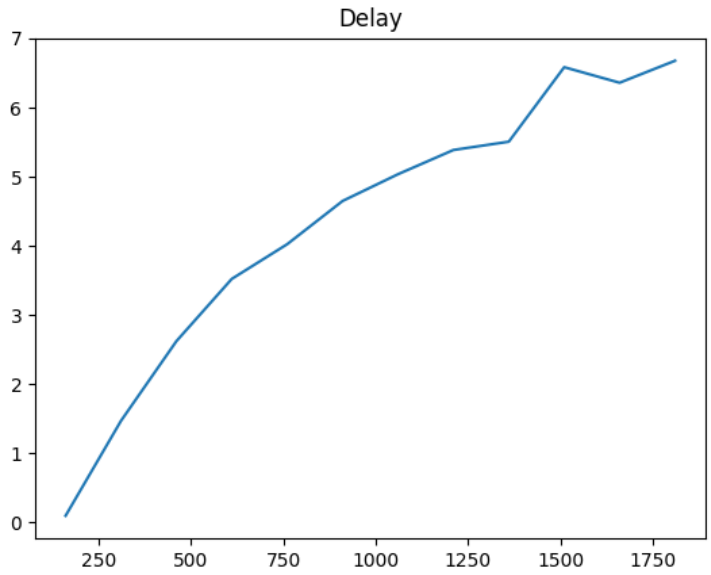


Figure 7: Change in Delay with Buffer Size

Performance testing consists of plotting the different parameters against the buffer size and analysing the end result. In this case, Delay, Packet Drop, Switch Packet Drop and Throughput are the parameters being plotted.

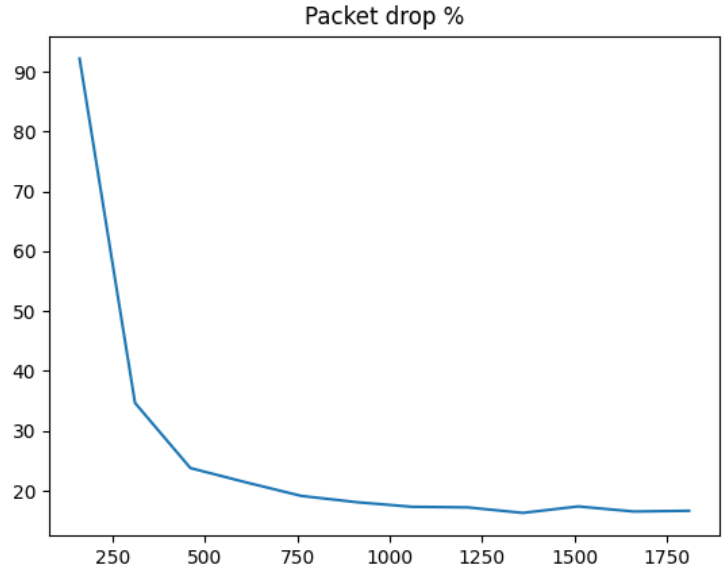


Figure 8: Change in Overall Packet Drop with Buffer Size

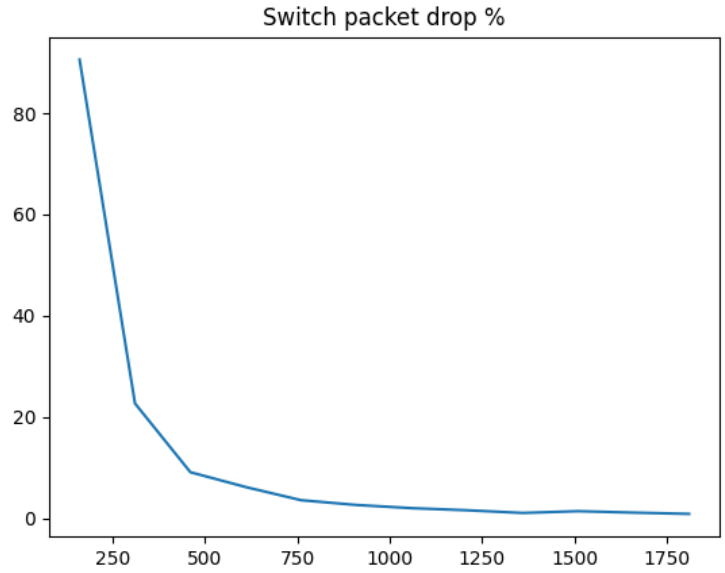


Figure 9: Change in Switch Packet Drop with Buffer Size

Beginning with the simple inferences, consider Figures 8 and 9. Total packet drop and packet drop show a very similar curve. This is because the number of packets being dropped reduces significantly with increase in buffer size. However, if one notices the y-axis of both the graphs, it can be seen that the Overall Packet Drop percentage stabilizes at around 20%, whereas the Switch packet drop stabilizes much closer to zero. This is because the network connections will show some packet loss regardless of how efficient and secure the switches are. Therefore, the reverse J curve shown in the graphs will remain the same except that it will be

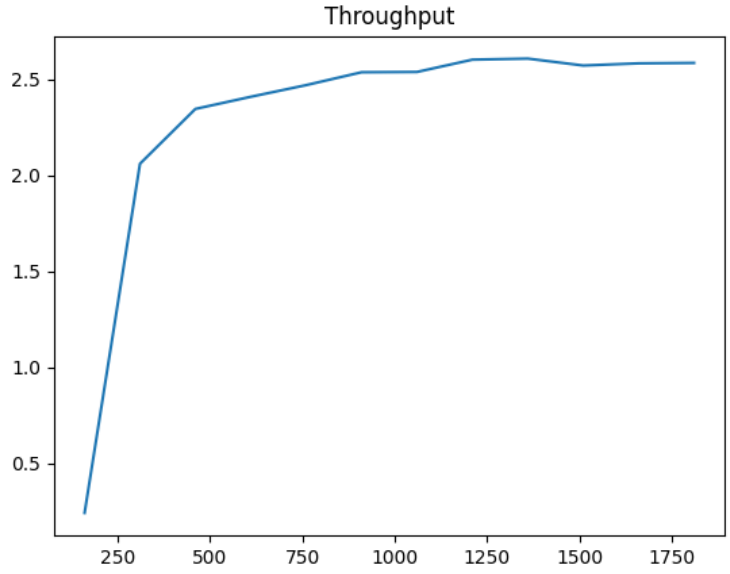


Figure 10: Change in Throughput with Buffer Size

shifted upwards in the Overall Packet Drop Graph. Considering them purely as performance parameters, there seems to be significant improvement in the packet drop percentage with the advent of an RL Agent.

The next inference would be a joint study of Figures 7 and 10. Despite an increase in buffer size, the throughput plateaus after reaching a maximum. This happens because the network is now able to successfully transmit almost all of the packets being generated. The average delay continues to increase because delay is calculated only with respect to the packets that reach the sink from the source. In that case, an increase in buffer size will cause an increase in waiting time regardless. This is where the RL Agent comes in. The delay will plateau over the time graph, as the buffer size is not increased by the RL Agent once the throughput plateaus.

6.4 Unit Testing

Unit Testing was conducted by providing different initial buffer sizes in order to settle on an optimum number to begin with. While comparing Figures 16 and 17, one can notice the erratic curve when one begins with a high initial buffer size. This is because the scale of the plot is different, and there was very low packet drop to begin with. However, on the off chance that the network traffic is low, starting off with a high buffer size is extremely wasteful.

6.5 Integration Testing

The project was tested for the successful integration of all modules. The modules worked together in synergy in order to provide the required output. The RL Agent worked with the simulation environment successfully and helped provide information to realise and plot the

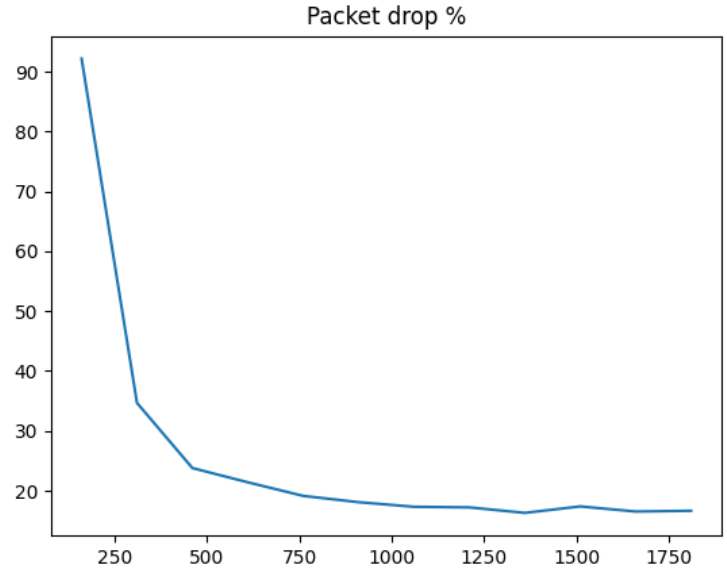


Figure 11: Overall Packet Drop with Low Initial Buffer Size

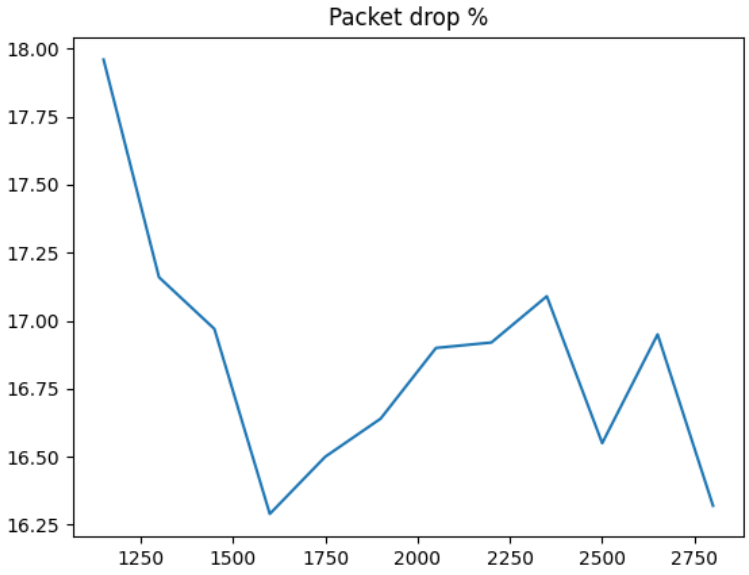


Figure 12: Overall Packet Drop with High Initial Buffer Size

results of the project.

6.6 System Testing

The project was uploaded on GitHub with clear instructions to run the program. The program's ability to run in different systems was remotely tested. Once the requirements are met, the code can be modified on any IDE and can run on any Operation System.

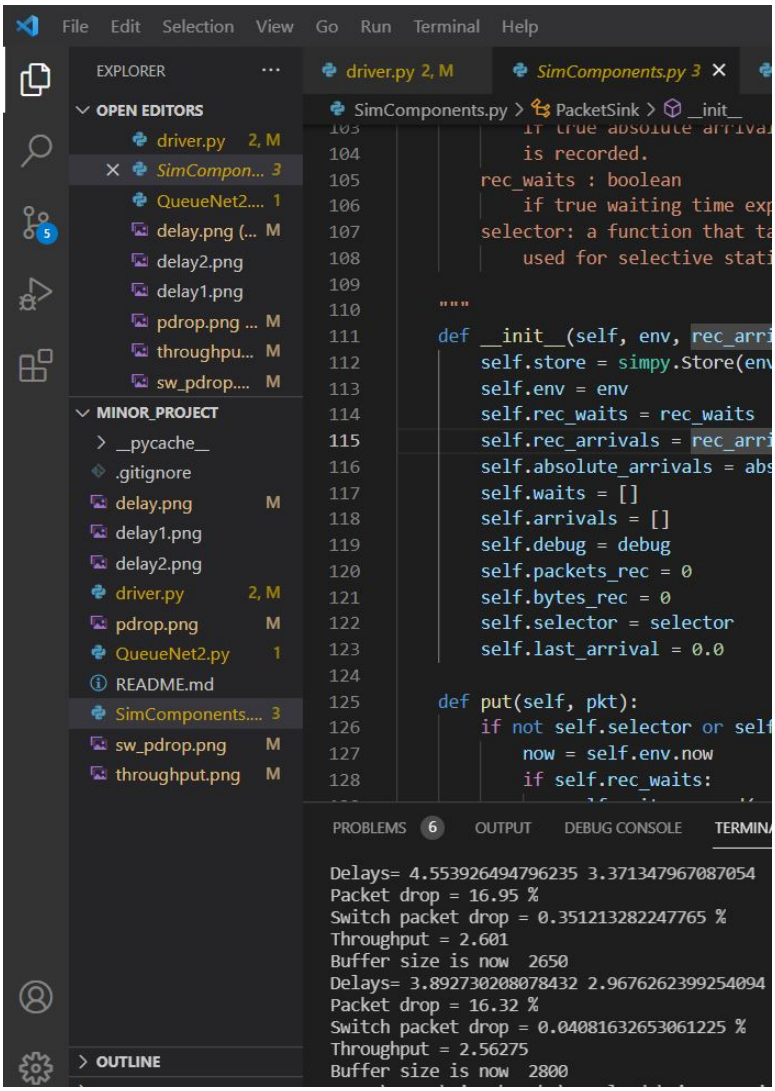


Figure 13: Project being edited on Visual Studio Code and run on PowerShell

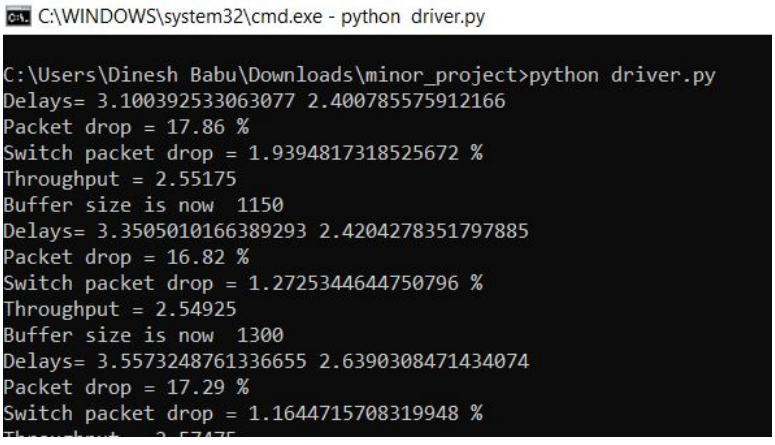


Figure 14: Project running on a different Windows Command Prompt

6.7 Summary

The various parameters for testing have been covered, and the results have been illustrated clearly in this chapter. Additionally, sufficient justification has been provided to the reader about the thought process behind the various decisions involved in the development process.

7 Conclusion and Future Enhancement

Through this project, the team was able to gain a lot of knowledge in various domains ranging from Computer Networks and Network Programming to Machine Learning. The team has also explored multiple platforms to make implementation decisions. This led to a vast expansion of the team's knowledge base as well.

Bringing back the attention to the topic of the project, this project clearly illustrates that it is a simple but effective idea to have the reward and penalty system for buffersizing. The project has produced tanglble output and has visualised it in order to illustrate the relationship that the performance metrics (Delay and Throughput) have with buffersizing.

7.1 Limitations of the Project

The main limitation of the project is the inability to visualise a network unlike in platforms like mininet which have their own ways of reprsentation in the form of GUI. Another limitation would be that the project has not accounted for TCP packets and will need to do so immediately. In addition to these, the initial buffer size had to be manually ascertained by trial and error. This can be improved in further iterations.

7.2 Future Enhancements

- The initial buffer size had to be manually ascertained by trial and error. This can be improved in further iterations.
- A big step up would be to switch to a platfor like MATLAB, Floodlights, Mininet, etc. One can also use NS3 like Sneha et. al., [10]
- Another enhancement would be to use TCP Packets

7.3 Summary

With this project, we conclude that the usage of a Reinforcement Learning agent to vary the buffer size of a switch in the data plane of an SDN would cause significant improvements in delay and throughput. This project currently simulated UDP communication, as the concept of acknowledgement packets in this simulation would have led to unnecessary complications. An improvement would be to switch to TCP.

References

[1] URL: https://www.grotto-networking.com/DiscreteEventPython.html.

[2] Nader Bouacida and Basem Shihada. "Practical and Dynamic Buffer Sizing Using LearnQueue". In: *IEEE Transactions on Mobile Computing* 18.8 (2019), pp. 1885–1897. DOI: 10.1109/tmc.2018.2868670.

[3] Enio Kaljic et al. "A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking". In: *IEEE Access* 7 (2019), pp. 47804–47840. DOI: 10.1109/access.2019.2910140.

[4] Minsu Kim. "Deep Reinforcement Learning based Active Queue Management for IoT Networks". In: (2021). DOI: 10.32920/ryerson.14649609.v1.

[5] Yohan Kim et al. "Buffer Management of Virtualized Network Slices for Quality-of-Service Satisfaction". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2018). DOI: 10.1109/nfv-sdn.2018.8725780.

[6] Fuliang Li et al. "Adopting SDN Switch Buffer: Benefits Analysis and Mechanism Design". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017). DOI: 10.1109/icdcs.2017.255.

[7] Yikai Lin et al. "Pausing and Resuming Network Flows using Programmable Buffers". In: *Proceedings of the Symposium on SDN Research* (2018). DOI: 10.1145/3185467.3185473.

[8] Ayan Mondal, Sudip Misra, and Ilora Maity. "Buffer Size Evaluation of OpenFlow Systems in Software-Defined Networks". In: *IEEE Systems Journal* 13.2 (2019), pp. 1359–1366. DOI: 10.1109/jsyst.2018.2820745.

[9] Deepak Singh et al. "Modelling Switches with Internal Buffering in Software-Defined Networks". In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)* (2018). DOI: 10.1109/icccn.2018.8487364.

[10] M. Sneha, Kotaro Kataoka, and G. Shobha. "BO-RL: Buffer Optimization in Data Plane Using Reinforcement Learning". In: *Advanced Information Networking and Applications Lecture Notes in Networks and Systems* (2021), pp. 355–369. DOI: 10.1007/978-3-030-75100-5_31.

8 Appendices

8.1 Appendix A: Screenshots

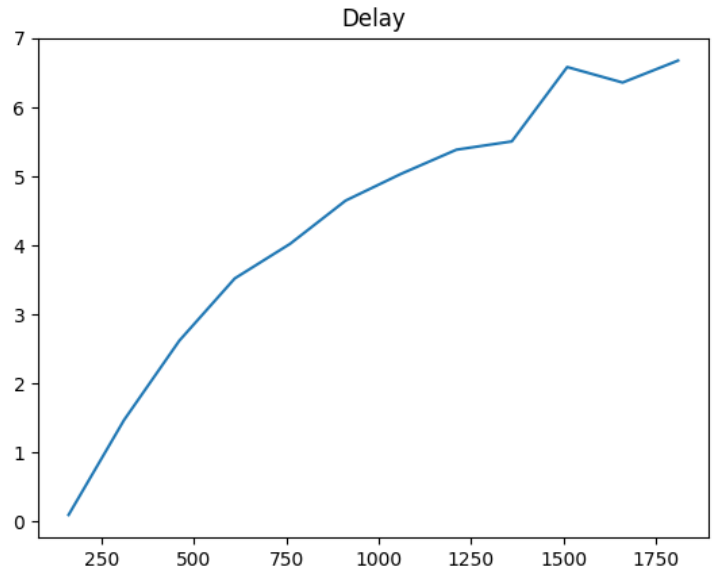


Figure 15: Change in Delay with Buffer Size

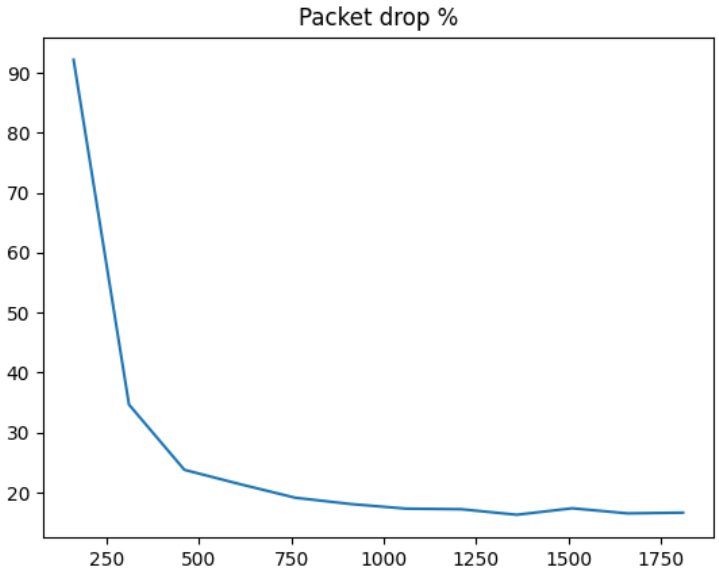


Figure 16: Change in Overall Packet Drop with Buffer Size

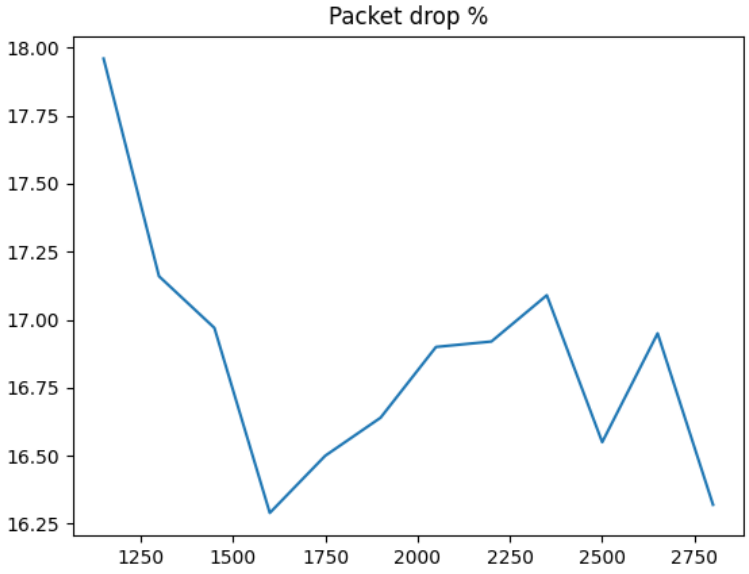


Figure 17: Overall Packet Drop with High Initial Buffer Size

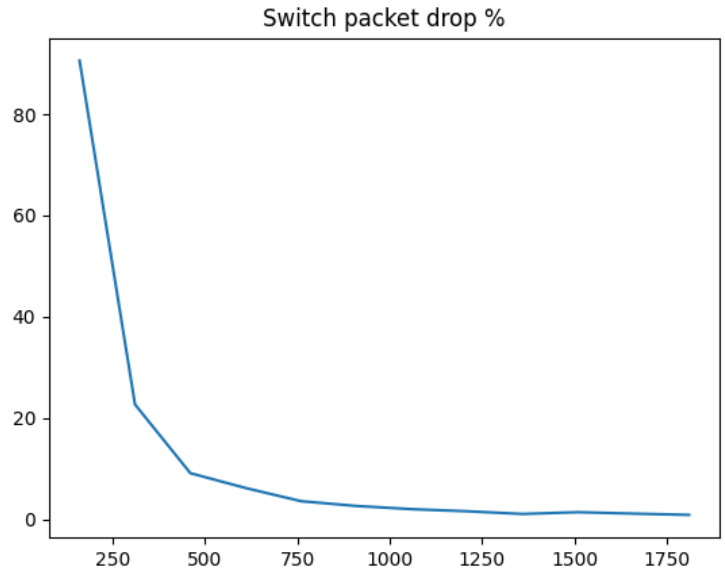


Figure 18: Change in Switch Packet Drop with Buffer Size

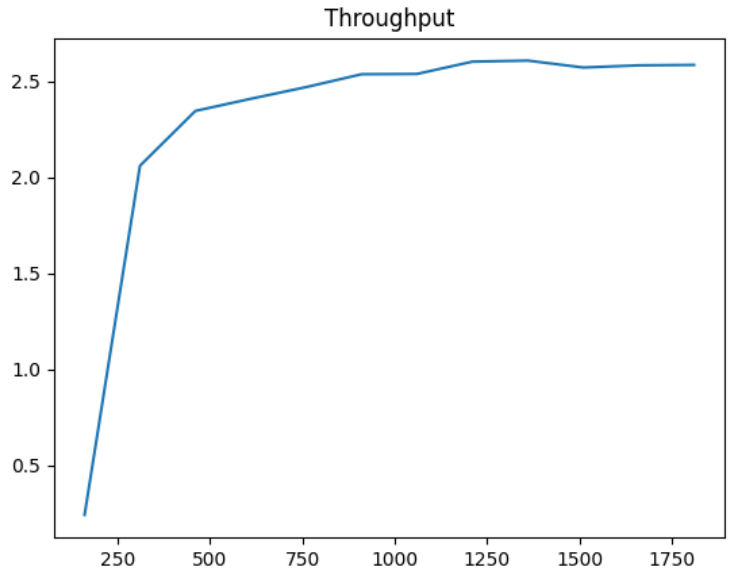


Figure 19: Change in Throughput with Buffer Size

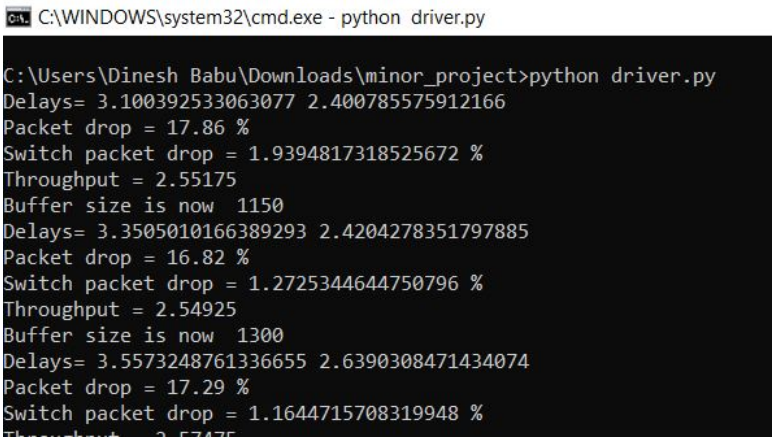


Figure 20: Project running on a different Windows Command Prompt

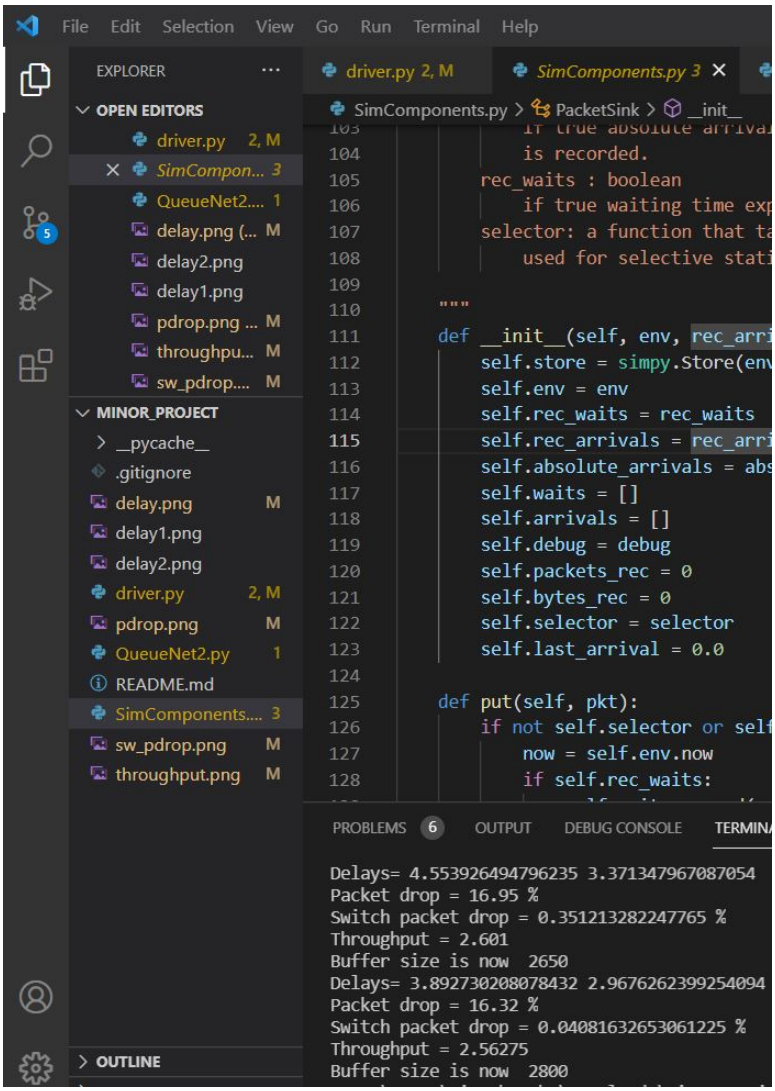


Figure 21: Project being edited on Visual Studio Code and run on PowerShell

8.2 Appendix B: Printout of the Base Paper.

BO-RL: Buffer Optimization in Data Plane using Reinforcement Learning

Sneha M
R V College of Engineering
sneham@rvce.edu.in

Kotaro Kataoka
Indian Institute of Technology Hyderabad
kotaro@cse.iith.ac.in

Shobha G
R V College of Engineering
shobhag@rvce.edu.in

***Abstract*—Fine tuning of the buffer size is well known technique to improve the latency and throughput in the network. However, it is difficult to achieve because the microscopic traffic pattern changes dynamically and affected by many factors in the network, and fine grained information to predict the optimum buffer size for the upcoming moment is difficult to calculate. To address this problem, this paper proposes a new approach, Buffer Optimization using Reinforcement Learning (BO-RL), which can dynamically adjust the buffer size of routers based on the observations on the network environment including routers and end devices. The proof of concept implementation was developed using NS-3, OpenAI Gym and TensorFlow to integrate the Reinforcement Learning agent and the router to dynamically adjust its buffer size. This paper reports the working of BO-RL, and the results of preliminary experiments in a network topology with limited number of nodes. Significant improvements are observed in the end to end delay and the average throughput by applying BO-RL on a router.**

***Keywords*–Reinforcement Learning, Network performance, Dynamic buffer optimization, Throughput, Latency**

I. Introduction

Low latency and high throughput are the global demands for any of modern networked services. Many optimization techniques have been explored to improve latency and throughput at the same time including Quality of Service (QoS)[1][2] transport layer protocol extensions [3][4][5] and traffic engineering in controlled environment such as Data Center Networks[6][7][8][9][10] dynamic buffer-size adjustment for packet forwarding in the data plane of the network has also been actively explored.[11][12][13][14]

Packet buffering in the data plane determines how many packets can wait before they are transmitted to the link or discarded. Consequently, it significantly affects the latency and packet loss rate caused by the devices in the data plane. However, algorithm based solutions[15][16] cannot exhibit their best performance if the traffic pattern goes out of their model. When a new destructive protocol emerges, such a protocol may introduce new factors and parameters that are not supported by those algorithm based solutions.

In order to achieve the resilience to the dynamic changes of traffic trend on the buffer size optimization in the data plane, this paper proposes Buffer Optimization using Reinforcement Learning (BO-RL). BO-RL uses Deep Q Learning to implement the Reinforcement Learning agent, which continuously makes the decision to change the buffer size based on the constant monitoring results of end to end latency and throughput in the network.

This paper developed a proof of concept (PoC) implementation using NS-3, OpenAI Gym, and TensorFlow to optimize the buffer size of one router in the network. The experimentation results showed the significant improvement on both latency and throughput, and left some lessons for both further improvement and real-world deployment. The main contributions of this research work are:

1) designing and implementing the practical use case of Reinforcement Learning to optimize the buffer size,
2) experimentally proving the effectiveness of the proposed approach in the small and controlled environment,
3) sharing the lessons for the real world deployment.

II. Related Work

The related work study is done in different ways to understand the existing solutions and its drawbacks. Broadly the study has been categorized as: **Machine Learning (ML) solutions which indirectly improve the performance parameters:** This category of the related work study refers to those research attempts which incorporates machine learning approach to solve different problem, but intern improved the performance parameters of our interest i.e. delay, throughput.

Quang Tran Anh Pham et.al[1] used Reinforcement Learning with DDPG(Deep Deterministic Policy Gradient) algorithm which takes traffic patterns as input and incorporates reward QoS into the reward function. This work explored a better routing configuration as the action of the agent, which also improves the latency. This work was implemented as a separate plane in SDN (Software Defined Networks). H. Yao et.al[3] proposed load balance routing schemes considering queue utilization (QU), which divide the routing process into three steps,namely the dimension reduction, the QU prediction using Neural Networks, and the load balance routing. This work was implemented in the control plane of SDN and compared with the other similar scheme. The authors had considered the worst throughput for analyzing the performance. They had assumed that if the worst throughput is higher, then the load balancing is better. The results showed a 20% improvements than the shortest routing scheme Bellman-Ford(BF).

Yiming Kong et.al[4] proposed TCP Congestion Control using Reinforcement Learning(RL) based Loss Predictors where the static buffer is altered manually at router to check different

scenarios. RL-TCP showed a 7-8% decrease in RTT and 9% increase in throughput on an extremely under buffered bottleneck link.

ML solutions which directly improve the performance parameters: This category of the related work study refers to those research attempts which incorporates machine learning approach to improve delay and throughput.

M. Elsayed and M. Erol-Kantarci in [17] proposed a Deep Reinforcement Learning based dynamic resource allocation algorithm to reduce the latency of devices offering mission-critical services for small cell networks. Both the base station and the end nodes were running the agent and results showed 30% improvement in the latency for dense scenarios with a 10% throughput reduction. N. N. Krishnan et.al[18] proposed a DRL(Deep Reinforcement Learning) framework to learn the channel assignment for the Distributed Multiple Input Multiple Output (D-MIMO) groups for WiFi network in order to maximize the user throughput performance. They achieved a 20% improvement in the user throughput performance compared to other solutions.

N. Bouacida and B. Shihada [11] proposed LearnQueue which is an AQM (Active Queue Management) technique using Machine Learning. It tunes the buffer size dynamically to decrease the time that packets spend in the queues and thus causing smaller delays. This is the base paper for our research work. The difference being the agent's algorithm and the choice of the observation parameters. There are different better algorithm exists other than Q-Learning which can significantly improve the performance of the agent. The main focus of our research is to reduce the end to end delay and see how learning the queue size will help achieve that. LearnQueue tries to reduce the queuing delay a packet experiences in the transmit buffers of the Access Point (AP) for different WiFi network settings. The agent was made to receive every 15ms the enqueue rate and the current queuing delay at the AP to get a reward/penalty. The authors observed from the experimental results that LearnQueue reduces queuing latency drastically while maintaining the similar throughput in most of the cases. Minsu Kim proposed [12] the Deep Reinforcement Learning (DRL) based AQM which selects a packet drop or non-drop action at the packet departure stage depending on the current state consisting of current queue length, dequeue rate, and queuing delay. After an action is selected, a reward is calculated based on the queuing delay and packet drop-rate. This is also a candidate research in comparison with the work proposed in this paper. Here the main difference is, our work proposes how to reduce the end to end delay rather than reducing the queuing delay. Dropping or non-dropping action of a packet is not taken by the agent rather it is left to the queuing discipline employed on the node.

In [11] or [12], the agent keeps learning and managing the transmit queues on every AP in the network. It is not highlighted on how to select the relevant AP for running the agent. If you run the agent on all the AP's in the network, then the cost and the complexities will increase. To take care of this issue, in our method the agent is made run on only those switches associated at the bottleneck link.

Recent advancements to improve the performance parameters which doesn't incorporate the ML algorithm in their solution: This category of the related work study refers to those research attempts which does not incorporate machine learning approach to improves the performance of delay and throughput.

S. Liu et.al [15] developed a low-latency application layer transport module based on node, which provides socket APIs to enable flow replication. The algorithm was implemented both on real and simulation environments. In this algorithm whenever an application initiates a flow, two TCP connections are started in different path to find out the RTTs. The one with less RTT is used to send the data. They observed a 69% and 57% 99.9%ile latency reduction respectively over linux stack TCP when implemented on a real test bed. The drawback of this work could be, the number of connections will increase two folded as the number of flows increase in the network. Homa[5] proposed by Behnam Montazeri et.al, is a transport layer connectionless protocol implemented using RPCs (Remote Procedure Call). It assigns priorities dynamically on receivers. It integrates the priorities with a receiver-driven flow control mechanism and uses controlled over commitment, where a receiver allows a few senders to transmit simultaneously.

Zonghui Li et.al. proposed Best Effort (BE) assisted communication protocol called as BEST TT protocol [16]. It forwards the copies of TT(time-triggered) frames with BE strategies, whichever copy is received at the end host first, is taken and the other frame is discarded hence the maximum time of frame delivery will be within the BE delivery time. This work optimized end to end latency but enlarged the jitter, the difference between the minimum and the maximum latency was increased. In [10] authors have proposed new data center protocol architecture (NDP). NDP impacts the whole stack including switch behavior, routing and a completely new transport protocol. NDP uses a receiver-driven transport protocol designed specifically to take advantage of multipath forwarding, packet trimming, and short switch queues. The goal at each step is first to minimize delay for short transfers, then to maximize throughput for larger transfers. The proposed architecture requires changes to the switch's queuing algorithm, multipath forwarding and modifications to the existing transport protocol TCP. It requires changes in many layers of the protocol stack, the paper also discusses the ways in which the NDP can be deployed on to DCNs having P4 switches coexisting with traditional TCP flows.

To the best of our knowledge, this paper exhibits the fist attempt to improve both of end-to-end delay and throughput by dynamically adjusting the buffer size of a packet forwarding device using ML.

III. System Design

A. The overall architecture

This paper proposes a system to improve end-to-end delay and throughput in a network by optimizing the buffer size

on the routers using Reinforcement Learning (RL) algorithm as shown in Fig. 1. The proposed architecture introduces RL Agent, which is an additional node in the network to run the RL operation. Network nodes and end hosts are extended to interact with RL Agent to allow it to give the instruction of buffer size adjustment on the network. There are four different steps in the RL operation: 1)Observing 2)Training & Learning 3)Taking the Action on the environment 4)Getting the reward. RL Agent observes the network state from both router and end hosts from the environment, takes actions based on the total reward earned for the previous action, gets the reward for current action.

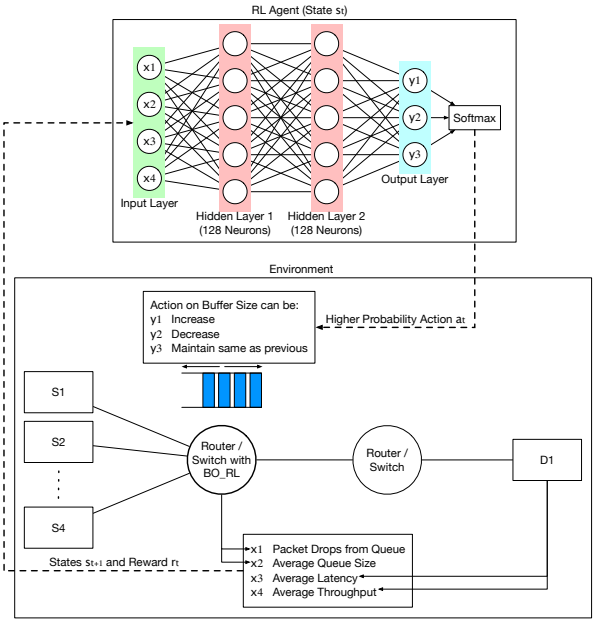


Fig. 1: Overall architecture of the proposed system

B. Formulating the state space s_t

Four parameters are considered to form a state space s_t=$\{x_1, x_2, x_3, x_4\}$ as shown in Fig. 1 The parameters x_1 = the number of packets dropped because of queue length and x_2 = average qsize at the router are observed from the switch/router. The parameters x_3 =average per packet delay and x_4 = average throughput are observed from the destination node.

1) Observing the router for the parameters x_1 and x_2: At every time step t the agent observes the network environment to read the average queue size and packets dropped due to queue size exceeded from the router. Here queue refers to the QueueDisc[19] of traffic control layer in Linux systems. Algorithm 1 is used for converting this network information from the router to agent's observation. *CheckQueueDiscSize* procedure is called every 0.1Millisecond to check what is the queue size. But the average queue size is fed back to the agent for every time step t only. Number of packets dropped because of queue was full at the time step t is also fed back with the average queue size to the agent's observation.

Algorithm 1: Function CheckQueueDiscSize

```
Result: Current queue size at R1
Pointer to queue at R1;
mygymenv is the gym object created for interacting
  with RL agent;
static int last_recorded_q_size_check_time=0;
static int avgQueueDiscSize,=0 checkTimes = 0;
newsize = get the predicted q size from the agent by
  mygymenv− >setq_size variable in number of
  packets;
curQsize = get the current qSize from Queue in
  number of packets;
curTime = get the current time in Millisecond;
avgQueueDiscSize += curQsize;
Increase the checkTimes by 1;
if curTime >= (last_recorded_q_size_check_time+2)
  then
    mygymenv− >cur_qsize =
      avgQueueDiscSize/checkTimes if newsize> 0
      newsize> curQsize then
        queue− >SetMaxSize(QueueSize(newsize))
          last_recorded_q_size_check_time = record the
          current time in Milliseconds;
        Get the dropped_packets stats from queue
          dropped due to LIMIT_EXCEEDED_DROP
          in number of packets;
        mygymenv− >dropPacket = dropped_packets;
    end
end
```

2) Observing the destination for the parameters x_3 and x_4: At every time step t the agent observes the network environment to read the average throughput and average per packet delay from the destination. Algorithm 2 is used for converting the network information from the destination to agent's observation. Whenever a packet is received at the destination node the *Delay_throughput* procedure is called, to record the delay experienced by that packet and the size of the packet (in bytes) received.

The information of the packet's transmission time is attached to the packet at the sender at the time of packet transmission, which is extracted at the destination node for calculating the delay. Average of the per packet delay and average throughput hence calculated in the time step t, is fed back to the agent's observation.

C. Choosing the algorithm for RL agent

Once the state space is formulated, the agent algorithm must be identified, which would best suit the requirements for BO-RL. The agent learning algorithms can be value-based, policy-based or model based [20]. The first two categories are model free algorithms and are suitable for networking research because the agent will not be having access to a model of the network environment.

BO-RL implements the DQN algorithm proposed by Mnihet.al.[22] with experience replay as the agent algorithm

Algorithm 2: Function Delay_throughput

```
Result: average delay and throughput
pointer to the packet received and DestIP;
mygymenv is the gym object created for interacting
  with RL agent;
static int totalReceivedbytes = 0;
static int last_recorded_thruput_time = 0 , pkt_cnt = 0;
if packet is the first packet then
    last_recorded_thruput_time = record the current
      time in Milliseconds;
end
if packet ! = 0 then
    pkt_size = get size of packet in bytes;
    now = current time in Milliseconds;
    pktTxTime = extract packet transmission time from
      SeqTsHeader attached at the sender in
      Milliseconds;
    perPktDelay = now – pktTxTime;
    totalDelay += perPktDelay;
    Increase pkt_cnt by 1;
    totalReceivedbytes += pkt_size;
    time_window += now in Milliseconds;
    if time_window >=
     (last_recorded_thruput_time+2) then
        delay = totalDelay / pkt_cnt in Millisecond;
        throughput = totalReceivedbytes /2 in
          Kilobytes;
        mygymenv− >thruput = throughput;
        mygymenv− >delay = delay;
        last_recorded_thruput_time = record the current
          time in Milliseconds;
        reset the variables totalDelay,
          totalReceivedbytes, pkt_cnt to 0;
    end
end
```

because; 1) DQN can accurately approximate the desired value function for the environments with large state space. Since we have network as the environment, it is impossible that the agent would observe a previously observed state 2) DQN's function approximation based value function can scale very well, which is a natural choice for more complicated large networks.

In the value based algorithm, the goal of the algorithm is to build a value function which eventually gets defined into a policy. The simplest algorithm in this category is Q-Learning[21]. Q-learning builds a lookup table of Q(s,a) values, with one entry for each state-action pair. Algorithm is well suited for the environments where the states and the actions are small and finite.

Q-learning algorithm cannot be used for BO-RL, because the environment in the proposed system is a network and is highly dynamic in nature. The state as defined in the section 3.B has 4 observation parameters and even a small change in one of the parameters will result in a different state, increasing the number of states to infinite. The Q-learning algorithm will start storing the Q-values for each such state-action pair into the Q-table and the convergence will be very slow. The memory requirement of the agent will also increase as the table size increases.

But the only limitation of the DQN is, because of the deep neural networks, which takes more time to train compared to QLearning. The DQN algorithm leverages the advantages of Deep Neural Networks(DNN) as a Qfunction that takes the state as input and outputs Qvalues for each possible action.

These Qvalues are optimized using mean squared error loss function (L), which is defined as in equation below:

$$L = \frac{1}{2}[r + \gamma max_{a'} Q(s^{'}, a^{'}) - Q(s, a)]^2$$

Where *r* is the total reward, *s* is current state, *a* is the action taken in current state, $s^{'}$ next state where the agent might move to by taking the action $a^{'}$ and γ is the discount factor to balance between immediate and future rewards: Every time, the network predicts: the Qvalues for all the actions, by feed forward pass for the current state s, calculates the $max_{a'} Q(s^{'}, a^{'})$ by doing a feed forward to state $s^{'}$, sets the Q-value for action a as $r + \gamma max_{a'} Q(s^{'}, a^{'})$, update the weights using the back propagation. The agent tries to minimize this error by learning, so that the difference between the predicted and the actual Qvalues is reduced. But convergence is still an issue.

To overcome this problem the authors in [22] have proposed two methods: 1) Experience replay 2)Target network. In experience replay method, instead of training the network and updating the weights for every transition, agent stores the last *N* experiences defined as $(s, a, r, s^{'})$in a replay memory. When the number of experiences in the replay memory is exceeding the minibatch size, agent selects the samples of the experiences randomly from the replay memory which will help in taking proper actions. This helps in breaking the strong correlation between consecutive samples and reduces the variance of the updates.

D. The working of RL agent and the network environment

The RL agent interacts with the Network environment *E* in a sequence of discrete steps=1,2,3.... At each time step *t*, the RL agent selects an action $a_t \epsilon A(s)$ from the set of action space on the given state $s_t \epsilon S$. Reward $r_t \epsilon R$ is received after taking the action a_t , and observes a new state s_{t+1}. BO-RL implements DQN algorithm with replay memory as the RL agent's algorithm.

At each time step, the state along with the reward are inputs to the agent. As an output, the agent returns one of the three possible actions (increase/ decrease/ maintain same Qsize). The higher probability action is taken on the router under observation in the network.

E. The process of training

The DNN is built using two hidden layers with 128 neurons for each layer. ReLU [23] is used for the hidden layers and softmax is used for the output layer as activation functions. The network weights are gradually learnt as the DNN gets

trained. Adam Optimizer [24], an optimization algorithm which is an extension to stochastic gradient descent is used to make the learning faster. Different hyper parameter of the RL agent are listed in Table II.

TABLE I: Hyper parameters of the RL agent

Parameter	Brief description
ϵ	Percentage you want to explore (selecting a random action) or exploit (selecting an action with max future reward)
Epsilon_min	Minimum ϵ value
Epsilon_decay	Rate at which ϵ is decreased from 1.0 to Epsilon_min
batch_size	Replay memory batch size
Gamma (γ)	Discount factor

For every time step t the RL agent learns the queue size in n number of episodes and each episode has m number of steps, so totally in $n \times m$ number of steps before an action is taken. Game over condition should be defined, so that the RL agent stops learning infinitely.

F. The Action a_t

The RL agent can take one of the actions(Increase/decrease/remain the same). The output layer of the DNN assigns probabilities to different actions and the Softmax function picks the action with higher probability. The selected action will be implemented back on the router in the network environment.

Every time the future actions which an RL agent takes are influenced by the reward which it had earned by taking the previous action. While choosing the action RL agent must decide whether to take the action randomly (exploration) or to take the same action which had got better reward in the past (exploit). This is a difficult choice and depends on the environment as well. Greedy policies can be used to balance between exploration and exploitation. BO-RL implements decaying ϵ – greedy policy to choose between exploration and exploitation while choosing the actions. After each time step t, ϵ is updated using the equation below:

$$\epsilon = \epsilon * Epsilon\_decay$$

G. The reward r_t

After the RL agent takes an action, it waits for the next state s_{t+1} to see the impact on the network. This impact is evaluated by a reward function to optimize the queue size further to achieve better delay and throughput. The reward can be on a single parameter or can be cumulative based on the requirement. A penalty will be given if the delay is higher and the throughput is decreased. The RL agent always tries to achieve higher rewards by which it automatically achieves better delay and throughput. The Gamma(γ), is used to balance between immediate and future reward; value near to 0.1 for future rewards and value near to 0.9 for immediate rewards. Since BO-RL is an online learning setting, only the immediate rewards have been considered for taking the action decisions.

IV. System Implementation

A. Implementation Details

BO-RL is implemented using four main components;Environment Network, OpenAIGym, ns3-gym and RL agent as shown in Fig.2.

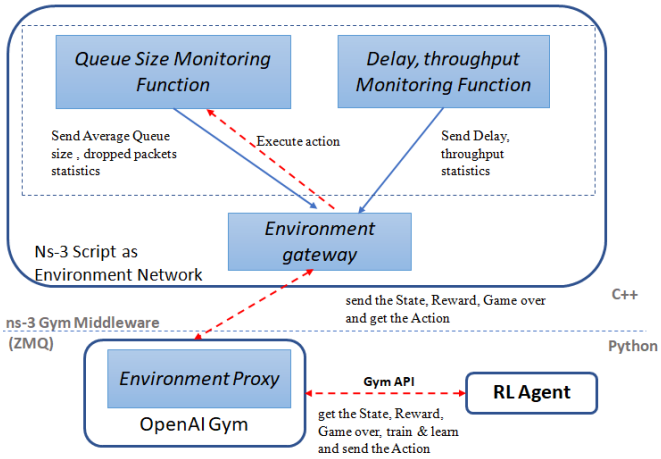


Fig. 2: System Block diagram

Environment Network: Environment Network is implemented using ns3 network simulator[25]. The simulation scenario script contains the internet topology and serves as the environment for the RL agent. The procedures(`Queue size Monitoring Function and Delay, throughput monitoring function`) in the script collects the observation parameter data as defined in section 3 and sends them to the RL agent through the `Environment gateway`. The procedure `Queue size Monitoring Function` executes the received action from the RL Agent. All the routers in the topology built, implement the DropTail queue as the device queue. The traffic control layer between the netdevice and the IP layer implements FIFO QueueDisc[26], which is manipulated by the RL Agent.

OpenAIGym: `gym` is a toolkit for developing RL algorithms. It is a collection of environments, which can be directly used to implement RL algorithms. Custom environments can also be added to this library according to the requirements of the problem definition. `gym` is independent of the structure of the RL agent implemented. The main purpose of `gym` is to have a standardized interface allowing to access the state and execute actions in the environment.

ns3-gym Middleware: It interconnects ns-3 simulator with OpenAI Gym framework. It provides communication between the components over the`ZMQ sockets` (ZeroMQ is a library used to implement messaging and communication between two applications) using `Protocol Buffers` (library for serialization of messages). ns3-gym has two parts; `Environment Gateway and Environment Proxy`. The `Environment Gateway` created by `OpenGymGateway` object resides inside the simulator and is responsible for:

1) Collecting environment state into structured numerical data as RL agent would expect using callback functions `GetObservationspace()`, `GetActionSpace()`, `GetObservation()`, `GetReward`, `GetGameOver()`
2) Translating the numerical actions received from the RL agent to proper function calls with proper arguments using the callback function `ExecuteActions()`

`Environment Proxy` created by the standard `Gym::make('ns3-v0')` receives environment state and exposes it towards the RL agent through the `Gym API` written in Python. The `Environment Proxy` translates the `gym` function calls into messages and sends them towards the `Environment Gateway` over `ZMQ socket`. **RL Agent:** The RL agent is implemented with Python 3.5 using TensorFlow [27] library for Deep Learning Networks.

B. Experimental Setup

Network Configuration and Flow Generation: In order to evaluate the effect of BO-RL, network topologies are created on NS3. Table 3 summarises the parameters and their values used for traffic generation and the RL agent implementation in the simulation.

TABLE II: Simulation Parameter Setting

Parameter	Values
Max size of the queue on all the routers	500 packets
Total size of data to transmit over TCP	1MB
Total Simulation time for testing	10s
Agents Observation Time interval `t`	2ms
Exploration rate ϵ	1.0
Epsilon_min	0.01
Epsilon_decay	0.999
batch_size	32
Discount factor Gamma(γ)	0.7
Learning rate	0.001
Total number of training episodes n	200
Total number of steps m in each episode n	100

TCP flows are generated using the BulkSend application of ns3 with randomized senders and receivers. The generation of flows follow a Poisson distribution while the start time follows an exponential distribution. The choice of initial queue size is important to allow the RL agent perform reasonably. The initial queue size was set to 300 packets that does not unnecessarily degrade the performance of the RL agent after performing preliminary experiments with different values between the minimum of 100 packets and maximum of 500 packets. If the initial buffer size is too small, it causes lots of packet drops and the throughput gets affected. If the initial queue size is too large, it affects the delay.

RL Agent Configuration: The possible actions to be taken by the RL agent are as follows:

1) increase the queue size by 1
2) decrease the queue size by 1
3) maintain the same queue size as previous

Initially the value of ϵ is 1.0 so that the RL agent will explore more by taking random actions and as the time passes the ϵ is eventually decreased by the epsilon decay factor to reach 0.01 where the RL agent tries to exploit by picking the actions from its stored experiences.

The game over condition is met, when the RL agent achieves 95% of the minimum possible delay (link delay). The below equation is used to check the game over condition:

$$delay{=}min\_delay * 1.05$$

The reward function is designed to give more priority to delay and throughput parameters than packet drops. RL agent gets cumulative rewards/penalty according to the conditions below:

1) if the *cur_delay*
   a) is lesser than the *prev_delay* a +ve reward
   b) is greater than the *prev_delay* a –ve reward
   c) is same as the *prev_delay* then a 0 reward
2) if the *cur_throughput*
   a) is lesser than the *prev_throughput* a +ve reward
   b) is greater than the *prev_throughput* a –ve reward
   c) is same as the *prev_throughput* then a 0 reward

V. Evaluation

We evaluate the BO-RL proof of concept implementation based on the latency and throughput of the end-to-end communication in NS-3 based testbed.

A. Choice of Reward function

Fig 3 plots the CDF of average reward earned by the RL Agent for 200 episodes during training.

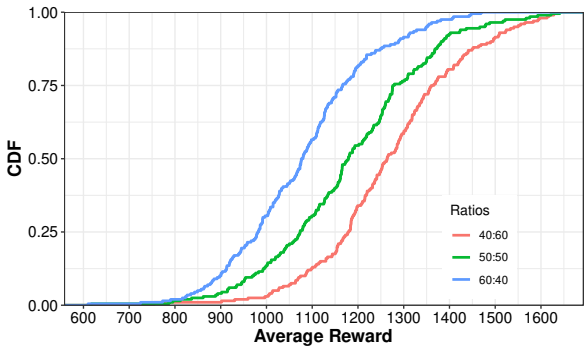


Fig. 3: CDF of Average Reward for 200 training episodes

The average reward and the standard deviation for different delay and throughput ratios were 1270 and 156 for 40:60 , 1183 and 168 for 50:50, 1079 and 148 for 60:40 respectively. Hence 40:60 reward ratio was considered in the conduction of all the experiments.

B. Buffer Optimization in Single Destination Environment

This experiment observes the baseline performance of the BO-RL RL by enabling it on Router 1 (R1) in the network given in Fig 4 on the basis of delay and throughput of the flows from multiple sources to a single destination.

We compare the results with R1 without BO-RL, which works as an ordinary router same as R2. In this setup for

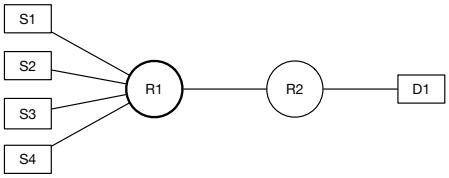


Fig. 4: The network configuration where the link between R1 and R2 is supposed to be congested due to the concurrent flows from 4 Sources to 1 Destination

enabling BO-RL, the RL agent receives the observation parameters from R1 and Destination 1 (D1). A total of 16 flows were generated by the sources to D1 to transfer 1 MB data in each flow.

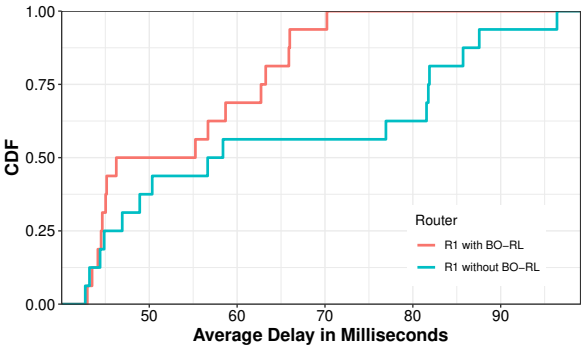


Fig. 5: Comparison of CDF End-to-end Average delay at D1 for the flows routed through R1 without BO-RL and R1 with BO-RL

Figure 5 plots the CDF of the average end-to-end delay of packets from the sources to D1 through the corresponding routers, and also shows that R1 with BO-RL successfully reduced the overall latency compared with R1 without it.

The average delay and its standard deviation for packets through R1 with BO-RL were 53.43 ms and 9.780 ms respectively, which out perform those of R1 without BO-RL 64.264ms and 19.347ms.The maximum latency with BO-RL was also significantly shorter than the case without BO-RL.

Fig 6 plots the CDF of the end-to-end throughput of flows observed in the same setup. The average throughput and the standard deviation were 2.215 Mbps and 1.534 Mbps with BO-RL, and 1.940 Mbps and 1.113 Mbps without BO-RL respectively.

Fig. 7(a) shows the queue size variation, and the number of packets dropped at R1 because of queue limits exceeded, and Fig. 7(b) the corresponding rewards gained by RL agent in the same test phase. Fig. 7a shows that the queue size is decreasing irrespective of packet drops observed at R1 since the throughput and delay is still improving and the reward gained is increasing as shown in Fig.7b. This is because of the choice of RL agent's reward function is more biased towards improving the delay and throughput rather than decreasing the packet drops.

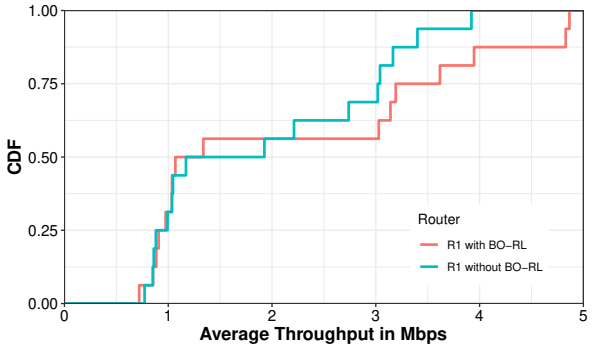
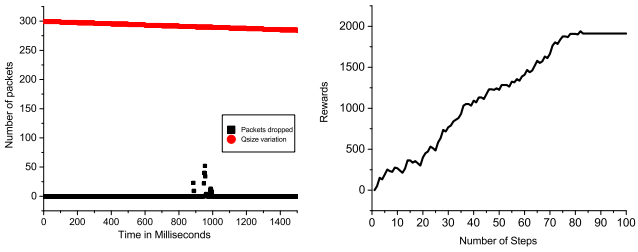


Fig. 6: Comparison of CDF End-to-end Average throughput at D1 for the flows routed through R1 without BO-RL and R1 with BO-RL



(a) (b)

Fig. 7: (a) Dynamic adjustment of queue size Vs number of packet dropped at R1 by RL agent, and (b) the corresponding reward gained

C. Performance of BO-RL with multiple Destinations and varying number of feedback sources

This evaluation item examines whether the performance of BO-RL improves if more receivers provide feedback to the RL agent. RL agent receives the observation parameters from R1 with BO-RL and more than one destinations,for dynamically optimizing the queue size at R1 as shown in Fig 8.

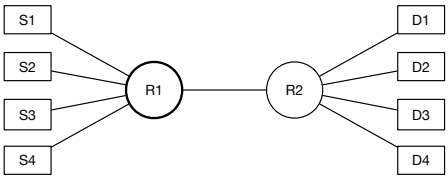


Fig. 8: The network configuration with 4 Sources and 4 Destinations

64 flows to transfer 1MB Data each were generated with randomized sources and destinations so that each destination receives up to 16 flows.

Tables III and Table IV compare the results of average end to end delay and throughput of flows between the R1 configuration with or without BO-RL.

TABLE III: Delay Analysis with and without BO-RL

	R1 without BO-RL	The number of feedback sources in addition to R1 with BO-RL			
		1	2	3	4
StdDev	3.558	3.667	1.912	2.543	1.395
Max	111.442	84.135	85.074	78.392	83.127
Min	97.966	69.436	77.735	69.089	78.247
Avg	105.084	74.055	81.641	74.584	81.354
Med	105.613	73.265	81.466	74.846	81.729

TABLE IV: Throughput Analysis with and without BO-RL

	R1 without BO-RL	The number of feedback sources in addition to R1 with BO-RL			
		1	2	3	4
StdDev	0.459	0.281	0.364	0.374	0.328
Max	1.937	1.048	1.454	1.657	1.430
Min	0.092	0.120	0.107	0.145	0.125
Avg	0.394	0.380	0.363	0.344	0.363
Med	0.230	0.262	0.230	0.254	0.253

In this configuration, the network congestion on the link between R1 and R2 is severer than the previous experiment due to the increased number of flows. The end-to-end delay has been significantly reduced regardless of the number of receivers to provide feedback to the RL agent. However, this experience did not clarify the relationship between the number of feedback receivers and the ratio of delay reduction. On the other hand, average throughput slightly degraded in this experiment.

D. Limitations and Potential Improvement

1) Network Topology and Traffic Pattern: The experiment was performed in a very small and simple network, and only TCP traffic was generated. While BO-RL shown the benefit of dynamic buffer optimization, it's benefit should also be explored in larger network typologies with variety of traffic patterns and applications. The number of BO-RL router can also be increased so that the buffer size optimization can happen in many parts in the network.

2) Number of Observation Parameters and Choice of RL Agent Algorithm: Our system used the following 4 parameters: the average queue size, the number of packets dropped from queue, the average throughput, and the average latency were used as observation parameters. Timestamp, which was used to measure the end-to-end per-packet delay, was obtained using the feature of NS3. In the real world networking scenario, precisely measuring one-way delay is not easy. However, there are some techniques [29][30][31] can be used to measure the one-way delay in the real world deployment. The observation parameters can be increased to feed more feedback to the RL agent, such as packet size, inter-packet arrival time, 5-tuple information (source/destination IP addresses and port numbers, protocol), etc. Several DQN extensions can also be explored for RL agent algorithm like Double DQN [32], Prioritized Experience Replay[33], Dueling DQN [34], Noisy Nets for Exploration [35].

3) Dependency on End User Feedback: In the current BO-RL implementation, the RL agent collects feedback from destination hosts in the network. While this mechanism works in NS-3 environment, it's not feasible in the real network. In order to complete the BO-RL operation without depending on the interaction with destination receivers, measurement, feedback collection and reward calculation need to happen inside the network. Software Defined Networking (SDN)[36], where SDN Controller can collect the observation parameters directly or indirectly from the switches, is a promising approach to reduce such dependency.

VI. Conclusion

This paper proposed BO-RL, a use case for Reinforcement Learning based technique on dynamic buffer optimization in data plane aiming for improving both end-to-end delay and throughput in the network. We conducted the preliminary evaluation of the PoC implementation of BO-RL, and observed that the use of BO-RL introduced the significant improvement to both end-to-end delay and throughput by 17% and 12% respectively compared with the router without BO-RL when the RL agent collects the feedback from the router and only one destination of the traffic. However, if the RL agent collects feedback from multiple destinations, while a minimum of 22.30% improvement was observed in the end to end delay, the throughput improvement was not so significant. Considering the size of network and traffic patterns used in the experiment setup, as well as the observation parameters using that the RL agent was developed, BO-RL still has room for the further improvement as discussed in Section V-D for future work.

References

[1] Quang Tran Anh Pham, Yassine Hadjadj-Aoul, Abdelkader Outtagarts. *Deep Reinforcement Learning based QoS-aware Routing in Knowledge-defined networking*. Qshine 2018 - 14th EAI International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness, Dec 2018, Ho Chi Minh City, Vietnam. pp.1-13. ffhal-01933970

[2] F. Chiariotti, S. Kucera, A. Zanella and H. Claussen, *Aalysis and Design of a Latency Control Protocol for Multi-Path Data Delivery With Pre-Defined QoS Guarantees* in IEEE/ACM Transactions on Networking, vol. 27, no. 3, pp. 1165-1178, June 2019, doi: 10.1109/TNET.2019.2911122.

[3] H. Yao, X. Yuan, P. Zhang, J. Wang, C. Jiang and M. Guizani, *Machine Learning Aided Load Balance Routing Scheme Considering Queue Utilization*, in IEEE Transactions on Vehicular Technology, vol. 68, no. 8, pp. 7987-7999, Aug. 2019.

[4] Yiming Kong, Hui Zang, and Xiaoli Ma. 2018. *Improving TCP Congestion Control with Machine Intelligence*, In NetAI'18: ACM SIGCOMM 2018 Workshop on Network Meets AI & ML , August 24, 2018, Budapest, Hungary.

[5] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout, *Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities*, In SIGCOMM, 2018. Complete version can be referred using https://arxiv.org/abs/1803.09615.

[6] M. Alizadeh et al., *Data center TCP (DCTCP)*, ACM SIGCOMM Comput. Commun. Rev., vol. 40, no. 4, pp. 63-74, 2010.

[7] B. Vamanan, J. Hasan, T. Vijaykumar,*Deadline-aware datacenter TCP (D 2 TCP)* , ACM SIGCOMM Comput. Commun. Rev., vol. 42, no. 4, pp. 115-126, 2012

[8] H. Wu, Z. Feng, C. Guo, Y. Zhang, *ICTCP: Incast Congestion Control for TCP*, Proc. ACM CoNEXT, November 2010.

[9] C. Wilson, H. Ballani, T. Karagiannis, A. Rowtron, *Better never than late: Meeting deadlines in datacenter networks*, Proc. ACM SIGCOMM Conf. (SIGCOMM), pp. 50-61, 2011.

[10] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik, *Re-architecting datacenter networks and stacks for low latency and high performance*, In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17). Association for Computing Machinery, New York, NY, USA, 29–42.

[11] N. Bouacida and B. Shihada, *Practical and Dynamic Buffer Sizing Using LearnQueue*, in IEEE Transactions on Mobile Computing, vol. 18, no. 8, pp. 1885-1897, 1 Aug. 2019.

[12] Kim, Minsu., *Deep Reinforcement Learning based Active Queue Management for IoT Networks* 10.13140/RG.2.2.24361.65126,2019.

[13] Radwan, Amr To, Hoang-Linh Hwang, won-Joo, *Optimal Control for Bufferbloat Queue Management Using Indirect Method with Parametric Optimization*, Scientific Programming, vol. 2016, Article ID 4180817, 10 pages, 2016. https://doi.org/10.1155/2016/4180817

[14] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik, *Re-architecting datacenter networks and stacks for low latency and high performance*. In Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017, 14 pages. https://doi.org/10.1145/3098822.3098825

[15] S. Liu, H. Xu, L. Liu, W. Bai, K. Chen and Z. Cai, *RepNet: Cutting Latency with Flow Replication in Data Center Networks*,in IEEE Transactions on Services Computing.doi: 10.1109/TSC.2018.2793250

[16] Zonghui Li, Hai Wan, Boxu Zhao, Yangdong Deng, and Ming Gu,*Dynamically Optimizing End-to-End Latency for Time-Triggered Networks*, In Proceedings of the ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies (NEAT'19). Association for Computing Machinery, New York, NY, USA, 36–42.

[17] M. Elsayed and M. Erol-Kantarci, *Deep Reinforcement Learning for Reducing Latency in Mission Critical Services*, 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, 2018, pp. 1-6.

[18] N. N. Krishnan, E. Torkildson, N. B. Mandayam, D. Raychaudhuri, E. Rantala and K. Doppler, *Optimizing Throughput Performance in Distributed MIMO Wi-Fi Networks Using Deep Reinforcement Learning*, in IEEE Transactions on Cognitive Communications and Networking.doi: 10.1109/TCCN.2019.2942917

[19] https://www.nsnam.org/doxygen/classns3_1_1_queue_disc.htmldetails

[20] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, *An introduction to deep reinforcement learning*, in CoRR, Dec. 2018.

[21] Watkins, C.J.C.H., Dayan, P. Q-learning. Mach Learn 8, 279–292 (1992).

[22] Mnih, V., Kavukcuoglu, K., Silver, D. et al. *Human-level control through deep reinforcement learning*, Nature 518, 529–533 (2015).

[23] C. E. Nwankpa, W. Ijomah, A. Gachagan, S. Marshall, *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*, 2018. arXiv:1811.03378 [cs.LG]

[24] Kingma, Diederik P. and Ba, Jimmy. *Adam: A Method for Stochastic Optimization* arXiv:1412.6980 [cs.LG], December 2014.

[25] https://www.nsnam.org/releases/ns-3-29/

[26] https://www.nsnam.org/docs/models/html/traffic-control-layer.html

[27] https://www.tensorflow.org/

[28] https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/train/AdamOptimizer

[29] A. Hernandez and E. Magana, *One-way Delay Measurement and Characterization*, in Third International Conference on Networking and Services. ICNS 2007, Athens, 2007 pp. 114-114. doi: 10.1109/ICNS.2007.87

[30] R. Kumar et al., *End-to-End Network Delay Guarantees for Real-Time Systems Using SDN*, 2017 IEEE Real-Time Systems Symposium (RTSS), Paris, 2017, pp. 231-242, doi: 10.1109/RTSS.2017.00029.

[31] Ting Zhang andBin Liu,*Exposing End-to-End Delay in Software-Defined Networking*, International Journal of Reconfigurable Computing, Volume 2019, Article ID 7363901, 12 pages https://doi.org/10.1155/2019/7363901

[32] Hado van Hasselt, Arthur Guez and David Silver, *Deep Reinforcement Learning with Double Q-learning*,arXiv:1509.06461 [cs.LG]

[33] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, *PRIORITIZED EXPERIENCE REPLAY*,arXiv:1511.05952v4 [cs.LG] 25 Feb 2016

[34] Ziyu Wang ,Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot and Nando de Freitas,*Dueling Network Architectures for Deep Reinforcement Learning*,arXiv:1511.06581v3 [cs.LG] 5 Apr 2016

[35] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg, *NOISY NETWORKS FOR EXPLORATION*, arXiv:1706.10295v3 [cs.LG] 9 Jul 2019.

[36] Hamid Farhady, HyunYong Lee, Akihiro Nakao, *Software-Defined Networking: A survey*, Computer Networks, Volume 81, 2015, Pages 79-95, ISSN 1389-1286, https://doi.org/10.1016/j.comnet.2015.02.014.