SANTA CATARINA STATE UNIVERSITY – UDESC
COLLEGE OF TECHNOLOGICAL SCIENCE – CCT
GRADUATE PROGRAM IN APPLIED COMPUTING – PPGCAP

GUSTAVO DIEL

APPLYING DATA CLASSIFICATION AND ACTOR-CRITIC REINFORCEMENT LEARNING TO NETWORK CONGESTION CONTROL ON SDN-BASED DATA CENTERS

JOINVILLE
2022

GUSTAVO DIEL

APPLYING DATA CLASSIFICATION AND ACTOR-CRITIC REINFORCEMENT LEARNING TO NETWORK CONGESTION CONTROL ON SDN-BASED DATA CENTERS

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the Master's degree in Applied Computing.

Supervisor: Guilherme Piêgas Koslovski

JOINVILLE
2022

Diel, Gustavo
Applying Data Classification and Actor-Critic Reinforcement Learning to Network Congestion Control on SDN-based Data Centers / Gustavo Diel. - Joinville, 2022.
80 p. : il. ; 30 cm.

Supervisor: Guilherme Piêgas Koslovski.
.
Dissertação (Mestrado) - Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2022.

1. Congestion control. 2. Data centers. 3. Software Defined Network. 4. Machine learning. I. Koslovski, Guilherme Piêgas . II. , . III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. IV. Applying Data Classification and Actor-Critic Reinforcement Learning to Network Congestion Control on SDN-based Data Centers.

GUSTAVO DIEL

APPLYING DATA CLASSIFICATION AND ACTOR-CRITIC REINFORCEMENT LEARNING TO NETWORK CONGESTION CONTROL ON SDN-BASED DATA CENTERS

Master thesis presented to the Graduate Program in Applied Computing of the College of Technological Science from the Santa Catarina State University, as a partial requisite for receiving the Master's degree in Applied Computing.

Supervisor: Guilherme Piêgas Koslovski

THESIS COMMITTEE:

Guilherme Piêgas Koslovski, P.h.D.
Santa Catarina State University

Members:

Charles Christian Miers, Ph.D.
Santa Catarina State University

Felipe Rodrigo de Souza, Ph.D.
Havan S. A.

Joinville, Jul 25th 2022

“No use holding on because nothing stays the same” (Klaypex)

RESUMO

Cada vez mais serviços buscam oferecer um alternativa tecnológica para seus clientes. Toda aplicação em larga escala precisa de um lugar para ser hospedado, e existem diversos serviços especializados em oferecer essas plataformas de hospedagem, tais como AWS e Google Cloud Platform. Com a explosão do crescimento do uso dessas plataformas, é importante que os recursos físicos sejam utilizados ao seu máximo. Contudo, as redes de computadores que fazem esses serviços funcionarem são recursos ubíquos e compartilhados, e para evitar a perda de dados e garantir a integridade dos serviços, o reenvio de pacotes de rede é um método comum utilizado. Dependendo do cenário, esse reenvio pode causar uma demora na transferência de pacotes, que entendem como perda de pacotes causando ainda mais reenvios, e assim há o congestionamento nas redes. O congestionamento pode ser em pequena escala, e apenas ocasionar atrasos nos serviços, ou ter maiores impactos, tais como a perda de dados e inoperabilidade dos serviços, efeitos que não são incomums nesses cenários. Os divesos algoritmos de controle de congestionamento existentes não tem conhecimento da rede, e assim não podem tomar decisões baseadas no que realmente está acontecendo, apenas no que os computadores que estão executando o algoritmo sabem. Apesar de existirem tentativas de algoritmos que tenham conhecimento de rede, normalmente dependem de hardware específico, resultando em baixa adoção no mercado. Especificamente, Software Defined Network (SDN) criou uma oportunidade para evitar congestionamento uma vez que o controlador centralizado pode obter dados em tempo real da rede, de todos os fluxos e *switches*. Porém, a quantidade de dados coletados é enorme, e algoritmos rápidos são uma necessidade. Neste sentido, o presente trabalho propõe **R**einforcement Learning- and **S**DN-aided **C**ongestion **A**voidance **T**ool (RSCAT), que usa classificação de dados para determinar se uma rede está congestionada e o modelo de ator-crítico para encontrar parametros melhores para conexões TCP. As análises experimentais mostraram que RSCAT pode diminuir o Flow Completion Time (FCT) dos algoritmos Data Center Transmission Control Protocol (DCTCP) e CUBIC em diversos casos, sem mudança de *software*, ou atualizações nas extremidades do Data Center (DC).

Palavras-chave: controle de congestionento, *data center*, *software defined network*, aprendizado de máquina

ABSTRACT

Every day, more and more services try to offer a technological alternative to their clients. Many motives are there, but each large scale application will need some hosting service, such as Amazon AWS and Google Cloud Platform, known as DC. With the explosive growth in usage of said platforms, it is crucial that the physical resources are fully utilized at their maximum capacity. Due to high usage, many safety measures are in places, such as high-capacity components, and algorithms like re-transmission of Transmission Control Protocol (TCP) packets to minimize any kind of loss or downtime. However, depending on the scenario, re-transmission may send too many packets in the network, increasing response time, and causing more packets to be re-transmitted and creating a positive feedback loop in the network. Network congestion may be on a small scale, causing only information delay for the services, but in worse cases, data losses or even service downtimes are not uncommon. Many congestion control algorithms out in the market have no network knowledge, and as such, they must act and rely only on the data provided by the computer they are running on. Some algorithms have tried to use data and state networks to improve the performance, but these solutions depend on specialized hardware that is not widely available, and thus, these solutions have low market adoption. Specifically, SDN created an opportunity to avoid congestion once the centralized controller can gather ongoing and historical information from all network switches and flows. However, the data gathered is enormous, and fast-computing algorithms are crucial for decision-making. In this sense, this work proposes **R**einforcement Learning- and **S**DN-aided **C**ongestion **A**voidance **T**ool (RSCAT), which uses data classification to determine if the network is congested and actor-critic reinforcement learning to find better TCP parameters. Using the data from the network itself, the algorithms have an advantage when congestion starts in the network, and it has to decide. Our experimental analysis shows RSCAT could decrease the FCT of TCP congestion controll algorithms like DCTCP and Cubic Binary Increase Congestion control (CUBIC) in several cases without requiring any software update on DC end-points.

Keywords: congestion control, data center, software defined network, machine learning

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Actor-Critic
ACK	Acknowledgment
ANN	Artificial Neural Network
API	Application Programming Interface
BBR	Bottleneck Bandwidth and Round-trip propagation time
BIC	Binary Increase Congestion control
CUBIC	Cubic Binary Increase Congestion control
CDF	Cumulative Distribution Function
CWND	Congestion Window
CWR	Congestion Window Reduced
CPU	Central Processing Unit
DC	Data Center
DCTCP	Data Center Transmission Control Protocol
DoS	Denial of Service
ECE	ECN-Congestion Experienced
ECN	Explicit Congestion Notification
ECT	ECN-Capable Transport
ETG	Empirical Traffic Generator
FCT	Flow Completion Time
ML	Machine Learning
MLP	Multilayer Perceptron
MSS	Maximum Segment Size
MDP	Markov Decision Process
POC	Proof Of Concept
OS	Operating System
QoS	Quality of Service
RED	Random Early Detection
RF	Random Forest
RL	Reinforcement Learning
RSCAT	**R**einforcement Learning- and **S**DN-aided **C**ongestion **A**voidance **T**ool

RTT	Round-Trip Time
RWND	Receiver Window
SDN	Software Defined Network
SEQ	Sequence
SYN	Synchronize
SSTRHESH	Slow Start Threshold
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol Internet Protocol
TDL	Temporal Difference Learning
VCC	Virtualized Congestion Control
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

Following the expansion of the Internet and computer networks as a whole, new data center-hosted applications are being created by the hour with new types of services and goals, all using the cloud and edge resources as a means of infrastructure and data transfer. As many data centers host many distinct kinds of applications and services with several connection patterns, network congestion is bound to happen independently of the hardware or network management policies. Solving this problem without acquiring more hardware is considered a real challenge for data center administrators (KANDULA et al., 2009; NOORMOHAMMADPOUR; RAGHAVENDRA, 2017).

Two main indicators commonly perceive congestion in Transmission Control Protocol (TCP)-based computer networks: first, the network state where the network packets overload the forwarding capacity causing packet loss, even when intermediate buffers are available; secondly, the congestion can be perceived when the packets' Round-Trip Time (RTT) passes a certain threshold and thus, the packet is empirically marked as lost and another one is sent in its place.

The congestion can happen in numerous ways, such as sending more packets into a server than it can handle, or too many simultaneous connections in the same network link, and many others (KANDULA et al., 2009). The problem is that when congestion is in place, it can lead to collateral damages to the services running on the network if left untreated. Such collateral damages can vary from data losses due to lost packets to degraded services performances, which can then inflict many other business-related problems on their own. So preventing congestion from happening is the first step, and treating it quickly is the second (NOORMOHAMMADPOUR; RAGHAVENDRA, 2017; KANDULA et al., 2009; LA; WALRAND; ANANTHARAM, 1999). A third step could also be to optimize the network itself, allowing for faster transfers or minimizing costs.

Currently, known TCP-based congestion control algorithms can be divided into two categories, local and network-aided. Local algorithms rely solemnly on the information provided by the computer they are running on. Without the information from the network, the algorithms have shown two main problems: they either act selfishly and try to take over the available bandwidth, or they have non-efficient decisions regarding whether or not the network is congested (LA; WALRAND; ANANTHARAM, 1999; FLOYD et al., 1999; STOICA, 2005; ZHANG et al., 2019; ABDELMONIEM; BENSAOU, 2021). On the other hand, network-aided algorithms require devices support in order to perform well, but they also have a configuration and optimization problems on their own (SURYAVANSHI, 2019; ALIZADEH et al., 2010; CRONKITE-RATCLIFF et al., 2016).

Currently, TCP relies on Receiver Window (RWND) and Congestion Window (CWND) for deciding the window size of data segments allowed to be transferred on each TCP flow. While RWND indicates the available capacity at the receiver, CWND gives insights on network congestion. Both are needed due to the original congestion avoidance and control premises (CHIU;

JAIN, 1989): given a distributed scenario in which users are not aware of other users' demands, all the users sharing the same bottleneck will receive the same feedback, eventually converging to the total load on the network. The premises hold even when end-points run different congestion control algorithms, as they are bounded by the same limitations, the network.

To disrupt this scenario, on Software Defined Network (SDN)-based Data Centers (DCs), centralized congestion control and avoidance mechanisms can be used. The SDN controller receives network data from the switches connected to it, and thus, has a comprehensive knowledge of past and ongoing flows on the network. Moreover, it can monitor the entire network creating a robust dataset. Essentially, this data can be used to figure out which DC paths are critical to applications, and are eventually suffering from network congestion, as well as to indicate what could be the most optimal configuration for each TCP flow. Indeed, if the connection is accurately configured (e.g., slow start parameters, RWND size, and congestion control thresholds), the congesting scenario can be avoided, and the total network usage optimized.

In fact, it is possible to have thousands of servers, and even more simultaneous connections, when dealing with contemporaneous DC networks. This means that the data gathered in the controller can be very voluminous, requiring the use of fast-computing algorithms to process all of it. This is where Machine Learning (ML) algorithms come in, as they will have been trained beforehand and can process large quantities of data almost in real-time due to high usage of parallel computing and mathematical optimizations (FU et al., 2020; BOUZIDI; OUTTAGARTS; LANGAR, 2019; BOUTABA et al., 2018). Indeed what makes the ML shine in the computing world is not the composing algorithms, but the data used to train them. While there are some differences between ML algorithms and each have , the main focus when dealing with ML is having meaningful data, and that is astonishingly abundant in DC's network. Among the ML categories and paradigms, we focus on supervised data classification and Reinforcement Learning (RL), which requires little computational effort to take action over continuous values (JAAKKOLA; JORDAN; SINGH, 1994; BHATNAGAR et al., 2009).

This work proposes the **R**einforcement Learning- and **S**DN-aided **C**ongestion **A**voidance **T**ool (RSCAT) tool for DCs, which is based on the capabilities offered by SDN (managed with OpenFlow 1.3 (FOUNDATION, 2012)) to gather network data and control TCP flows. We use ML techniques such as data classification to determine if the network is or is not congested, and the Actor-Critic (AC) RL method to find the appropriate RWND configuration for ongoing TCP packets, as this technique has been efficiently applied to congestion control (HE et al., 2016; CRONKITE-RATCLIFF et al., 2016) without requiring updates on legacy hosts, operating systems and modules. RSCAT is designed to act on elephant flows (e.g., backup, synchronization traffic and large file transfers) while not negatively impacting the mice ones (e.g., network control and time-sensitive data). The experimental analysis with an SDN-based DC demonstrates that the RSCAT prototype decreased the average Flow Completion Time (FCT) when compared to traditional end-to-end and network-assisted algorithms, indicating the potential application of AC RL to avoid congestion on TCP-based networks.

1.1 OBJECTIVES

General objective: The main objective of the present work is to combine the data SDN provides with the computational power of ML algorithms such as Random Forest (RF) and AC to data knowledge to enhance current TCP-based congestion control algorithms.

Specific objectives: The main objectives in this work are:

- Study the congestion control problem;
- Study congestion control algorithms for the TCP protocol;
- Study congestion control algorithms for TCP-based DCs;
- Study new techniques and how to use them to enhance TCP congestion control (such as SDN, ML, and other);
- Study the AC and RF ML algorithms;
- Model a proposal with the new knowledge learned and proposed an implementation;
- Check the proposal's viability with manual and specific preliminary tests;
- Implement the proposal in a Proof Of Concept (POC); and
- Conduct an experimental analysis.

1.2 METHODOLOGY

The method adopted for the present master thesis is classified by its nature, objectives and approaches. Considering that the background research on the specific fields is a crucial step towards the objective of this work, and based on the knowledge gained from such research and other related works, this work can be classified as a referenced search. The first part of this thesis, the background, is composed of the definition of the core concepts related to this thesis' field of research, congestion control, SDN and ML, with the objective of improving the author's knowledge of such subjects. Next, it is reviewed some related works found in the literature to understand the current state-of-the-art congestion control mechanisms and what technologies are being used for this objective. And finally, it is proposed the mechanism created by the author and make some preliminary experimental tests to validate the proposal's functionality, along with an analysis of the results.

1.3 ORGANIZATION

This thesis is structured in the following manner: In Chapter 2 it is presented some background about the core concepts of this thesis, congestion control and its algorithms; congestion

control in DC; and SDN and how it can be used to aid congestion control. In Chapter 3, is presented the concept of ML, RL and a specific technique called AC, and how and why it can be used alongside SDN for congestion control. In Chapter 4, it is presented this master thesis' proposal, the RSCAT. It also presents related works that have been of great importance to the construction of this proposal. Chapter 5 brings some experiments and their results together with the analysis of said results. Finally, in Chapter 6 it is presented the final conclusions of this thesis, ranging from what are the main contributions of this thesis, what was done and possible future projects.

2 BACKGROUND

This chapter contains the core concepts which are the basis for this proposal's full understanding. Section 2.1 presents an overview of the definition and some key concepts of congestion in TCP-based networks. It also presents some congestion control algorithms built upon the TCP and its main concepts. On Section 2.2 is presented additional information on how data centers may control network congestion, along with other algorithms of other network layers and how they work. Section 2.3 presents the paradigm of SDN, its ideas, concepts and how to use it to control network congestion. Finally, on Section 2.4, it is presented the partial considerations for this chapter and some information about the problems faced in controlling network congestion.

2.1 CONGESTION CONTROL IN TCP-BASED NETWORKS

Today's computer networks serve as the commonplace for distributing information, sending and receiving binary data at the speed of light so that meaningful information is passed from one device to another. In an utopia, the network operations would achieve 100% of their maximum bandwidth all the time, bounded only by the network infrastructure and contract. But that is not the case. In the real world, where innumerous connections are sending and receiving data every single moment, the network protocols and infrastructure cannot handle such demand all the time, and that is when the users (or applications) perceive the slowdown of their requested operations (NAGLE, 1984).

What is really happening behind the scene is called network congestion and may have multiple causes, but the underlying effects are clear. Sometimes there may be too many connections from many distinct sources in the same network infrastructure, and when the throughput of the physical environment starts to become the bottleneck, the packets will then start competing against each other for bandwidth and network resources, causing the increase of the RTT of said packets. RTT is a measured attribute of a packet in a TCP connection. It is the time taken for each packet once they reach their destination and the destination returns an Acknowledgment (ACK) packet to the source, indicating that the initial packet was received (STEVENS, 1997).

As this value increases and the RTT passes a certain threshold, the source of the packet will then assume that something happened to the packet and it was not successfully received in the destination. The source is then expected to re-send the same packet over and over, at predetermined time intervals, until it has exhausted its retries or the message has been successfully received and a ACK packet returned. This mechanism is called re-transmission and it is placed in order to prevent data loss in a TCP connection (STEVENS, 1997).

If the network state stays degraded for long periods of time, the duplicated packets start to overload the connection, increasing the RTT of other connections and causing a non-stop avalanche of packets and re-transmission packets. When this happens, we have a heavily congested network that has reached a critical state (NAGLE, 1984; KANDULA et al., 2009;

HANDLEY, 2006).

The re-transmission mechanism usually prevents the connection from data loss, but even with the best algorithms, it is still possible the RTT of the packets will rise faster than the connection's capable of delivering it's packets. When this congestion starts, the re-transmission packets will start to fill up not only the network itself, but also the buffers of their sources and destinations. With the buffers at full capacity, packets will no longer have space to be processed or wait and must then be dropped. This is the worst-case scenario for a network congestion, and what worsens the situation is that this condition is stable, meaning that as the network is congested, the RTT stays high, generating more duplicated packets, and thus, the network keeps congested (NAGLE, 1984).

There are many reasons for congestion to happen in a network, and it starts when something disturbs the RTT of current connections in the network. Reasons like bad network configuration, heavy network traffic and multiple connections simultaneously are not rare in today's networks (NAGLE, 1984; KANDULA et al., 2009; STEVENS, 1997). The resulting effects of the congestion may vary on a case-by-case basis, but usually, it means degraded performance and, if more serious, broken connections and loss of data. In order to minimize congestion's effect on the network or even avoid it, congestion control methods have been developed in the many network layers that there are. For the TCP protocol, there are a large amount of algorithms that function as congestion control algorithms. The major components and ideas shared by all algorithms are revised in the following sections.

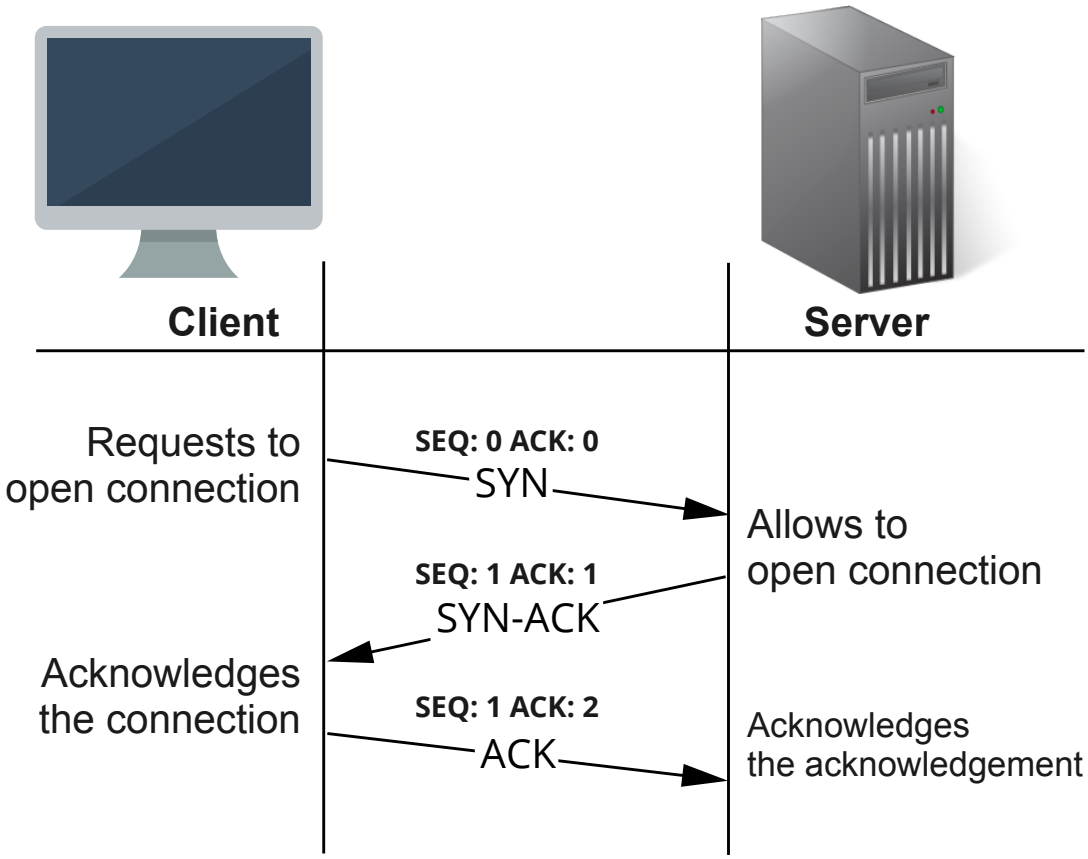
2.1.1 Receiver Window

The inner workings of TCP's congestion control algorithms start at the connection. TCP uses a Triple Handshake to initiate the connection, and it starts with the source sending a Synchronize (SYN) packet requesting to open a connection. Then, the destination responds with an SYN-ACK packet, confirming and initiating the connection. Once this SYN-ACK packet reaches the source, the source answers with another ACK and starts sending the desired data, as illustrated in Figure 1 (STEVENS, 1997).

During the handshake, the TCP extracts the RWND size for that connection from the ACK packets. In short, it is the size of the buffer available for the recipient to store and process that connection's packets. This information is constantly updated and written in the TCP's packets header, in the *window size* header field, and the source uses it to check how much data it can send before filling up the entire destination's buffer (STEVENS, 1997).

Another important field in the header is the sequence number. This 32 bits number is located in the header and keeps track of every single sent byte, and works by incrementing itself following the sent data amount. This data is important because it acts as an identifier for the packet. Once a packet reaches its destination, the destination will then send back the ACK packet with the same sequence number plus one, identifying that it has received the packet with that sequence number and is ready for the next packet. As in the example in the Figure 1, the Client

Figure 1 – Illustration of triple handshake in action in a simple TCP server to client connection. Source: author.



starts with the sequence number (relative) as 0. The server acknowledges that, and returns and ACK of $0+1$, which is the last Sequence (SEQ) plus 1, informing that it has received the full data and is ready for the packet with SEQ equal to 1. The client then sends a new packet with the sequence number 1, the one the server is expecting.

2.1.2 Slow Start and Congestion Avoidance

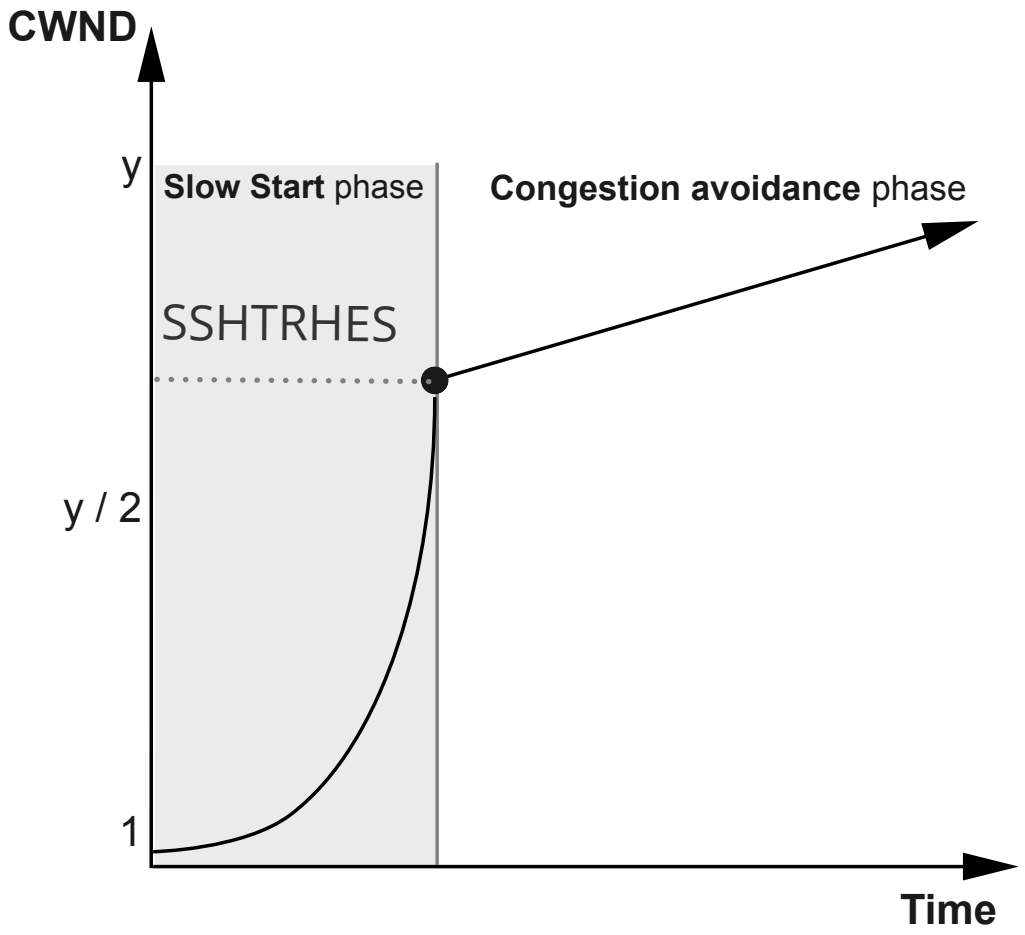
While the destination is responsible for configuring the RWND, the source is responsible for defining another variable, the CWND. The CWND value starts as one Maximum Segment Size (MSS). The segment is the limit size of a packet, while MSS is the maximum segment size defined by the Transmission Control Protocol Internet Protocol (TCP/IP) mechanism stored in the source's memory.

The CWND works by limiting how large the data should be sent in each TCP packet, in bytes, and finding the correct initial values of CWND is the goal of the slow start algorithm. Starting at the MSS value, the algorithm increases the value at each successful packet sent and response (from an ACK), at an exponential rate (JACOBSON, 1988). First, it is 1 MSS, then 2 MSS, then 4, and so on. It keeps increasing until it either finds a loss (which may be a duplicated ACK or an RTT above the threshold) or the RWND becomes a limiting factor. In these cases,

the CWND value decreases. The fallback value depends on the specific version of the algorithm. The original algorithm would be set back to one MSS value (STEVENS, 1997; NAGLE, 1984; JACOBSON, 1988), while other implementations have less aggressive methods (HA; RHEE; XU, 2008; CARDWELL et al., 2016). It is worth to mention that the slow start algorithm does not operate during the entire connection. After the CWND passes a value called Slow Start Threshold (SSTRHESH), the slow start will give place to the congestion avoidance algorithm, which, as the name suggests, allows the connection to avoid congestion. The initial value of SSTRHESH is half of CWND.

In congestion avoidance mode, the CWND will no longer increase exponentially. Some algorithms have a linear growing (following the formula $segSize^2/cwnd$, being $segSize$ the size of the segment provided by the Operating System (OS), in bytes) (STEVENS, 1997; NAGLE, 1984; HANDLEY, 2006; JACOBSON, 1988), while other proposals are based on specific functions (HA; RHEE; XU, 2008). Both slow start and congestion avoidance algorithms are illustrated in Figure 2.

Figure 2 – Illustration of a generic slow start and congestion avoidance implementations.
Source: author.



Both algorithms (slow start and congestion control) achieve two benefits for the TCP data flows (STEVENS, 1997; NAGLE, 1984; JACOBSON, 1988). First, it allows the connection

itself to reach the largest packet size possible without unnecessary packet loss due to grater-than-supported sizes, so no wasted bytes are needed to be processed. It also avoids instantaneous congestion with the old connections taking place in the network, as they will have time to adjust, and the new one will have a CWND value that corresponds to the network's state.

2.1.3 Fast Retransmit

By default, the way TCP acknowledgment works is that when a packet is received by the destination, it returns an ACK packet specifying the last in-order segment index of the connection. This means that, for example, if the destination receives the packets 1 through 6, but the packet 3 is lost, the ACKs returned will be 1, 2, 2, 2, 2. This means that it is possible for the source to receive duplicated ACKs during the transmission, and it is assumed that something has happened in the network. In this sense, what the Fast Retransmit algorithm does is, whenever three duplicated ACKs are received, it assumes that the next in-order packet was lost (in the example above, it had to be assumed that the packet 3 was lost). When this happens, the source retransmits the potentially lost packets instead of waiting for the retransmission timeout, which skips the delay in retransmitting the missing packets (STEVENS, 1997).

In addition, other TCP implementations use the Fast Recovery algorithm right after the Fast Retransmit in order to quickly recover the transmission window, skipping the Slow Start algorithm altogether. It accomplishes this by setting both *ssthresh* and CWND to half of the current CWND.

2.1.4 TCP Congestion Control Algorithms

Since its conception, TCP's congestion control has been undergoing changes, and multiple versions have been created and are available to use. Among the vast catalog of algorithms available, there are some that stand out and are related to the present work. Besides reviewing the fundamentals, a brief discussion on the drawbacks of each variant is performed:

- **Tahoe**

  TCP Tahoe, released in TCP version *4.3BSD-Tahoe*, is the first to use the Fast Retransmit algorithm and was widely available due to *4.3BSD Network Release 1* (JACOBSON, 1988). The first problem with Tahoe is that it takes at least the entire timeout time to detect packet loss and acknowledge it. In addition, another drawback is that it follows a "*go back n*" approach, which is that whenever a loss is detected, it does not send the immediate ACKs, but cumulative ACKs, and thus, it loses everything that was sent after the loss packet, which brings a significant slowdown for high bandwidth networks, especially with high congestion levels (STOICA, 2005).

- **Reno**

After TCP version *4.3BSD-Reno*, Tahoe was updated to Reno and with it, came the Fast Recovery algorithm and moreover, this version stopped using *go back n* (JACOBSON, 1990; STOICA, 2005). However, a problem that comes with Reno is its slowness in detecting packet loss, as it needs the prior packet to be fully acknowledged in order to detect the loss of the current packet. In short, it only detects one loss at a time, so under some circumstances it behaves similarly to Tahoe, especially in high bandwidth networks, even more under congestion (JACOBSON, 1990; STOICA, 2005). Another problem is when the window is small enough Reno would never receive the duplicated ACKs to activate the Fast Recovery and Fast Retransmit methods. On both Reno and Tahoe, the Slow Start algorithm reduces the CWND to 1 MSS when congestion is detected.

- **New Reno**

  The New Reno version aims to improve TCP Reno's Fast Retransmit and Fast Recover. It corrects the problem in Reno by being able to detect multiple packet losses simultaneously. The improvements from New Reno are that in the Fast Retransmit phase, it sends every packet that might have been affected by the loss, so it manages the resources available more efficiently, not waiting as long to resend the packets as Reno (FLOYD et al., 1999; STOICA, 2005). However, New Reno takes a full RTT to acknowledge each packet loss, as only after the ACK for the first re-transmitted packet was received can it deduce another packet was lost.

- **CUBIC**

  TCP Cubic Binary Increase Congestion control (CUBIC) is an improved version of Binary Increase Congestion control (BIC), and its protocol is to modify the linear window growth function to be cubic instead of the default linear one. It also turns the congestion control algorithm independent of the RTT (and thus, from the ACK, aggressively increasing the window size and slowing down when reaching the saturation point. The innovations allow CUBIC to be very efficient in high bandwidth and high latency networks (HA; RHEE; XU, 2008), being selected as the default TCP congestion control algorithm on MS-Windows Server 2019 version 1709, Windows 10 build 1709 and GNU/Linux since 2.6.19.

- **Vegas**

  TCP Vegas is based on Reno, but with a proactive approach instead and with the Internet in mind (BRAKMO; PETERSON, 1995). The biggest change is that it keeps track of each segment sent, and calculates an expected RTT for it. Based on these calculations, it proactively deduces a packet loss or congestion and starts the new re-transmission and congestion avoidance methods as needed (BRAKMO; PETERSON, 1995; BRAKMO; O'MALLEY; PETERSON, 1994). One problem with Vegas is on multiple routes: it is possible that some variances on the RTT might occur, and these variances could

trigger undesired congestion avoidance or even re-transmission actions (LA; WALRAND; ANANTHARAM, 1999). In addition, another problem is intrinsic using the estimation of propagation delay as a variable, and that it depends on the accuracy of its calculations to not over or underestimate its expectancy.

- **Bottleneck Bandwidth and Round-trip propagation time (BBR)**

  BBR is a congestion control algorithm developed by Google, in 2016. BBR is similar to Vegas in the sense that it is not a loss-based algorithm (which uses packet loss to detect congestion), but instead, it uses a model that takes in the maximum bandwidth and round-trip time of the last data packet to detect any congestion. It uses an internal clock and memory to record the amount of data delivered over time and how much of that data was acknowledged (CARDWELL et al., 2017).

  BBR has shown that it can improve the throughput in certain DC scenarios, but has also shown that it has queuing delays, lack of fairness when sharing the network with other algorithms, packet loss, and poor performance over cellular connections (HOCK; BLESS; ZITTERBART, 2017; ABBASLOO et al., 2018). It has a second version, BBRv2, which corrects its fairness issues, but delivers a lower throughput (GOMEZ et al., 2020).

It is important to remember that all of these algorithms are based on an end-to-end workflow, and thus, they can only operate on what the destination and mainly the source can detect, which are ACKs, RTT, and timeouts. In short, all algorithms are tailored to Internet traffic. The Table 1 presents a comparison between said algorithms and some key topics, like the feedback type and main benefits.

Table 1 – Comparison of popular TCP congestion control algorithms, comparing their intended target, feedback data source, main, required changes, year of proposal and main benefit

	Target	**Feedback Type**	**Required component change**	**Proposal year**	**Main benefit**
Tahoe	General	Loss	None	1988	Fast Retransmit
Reno	General	Loss	None	1990	Fast Recovery
New Reno	General	Loss	None	1999	Improvements
CUBIC	General	Loss	Sender	2005	Cubic window growth
Vegas	General	Delay	Sender	1994	Less losses with proactive changes
BBR	Data Center	Delay	Sender	2016	Buffer optimization and enhanced proactive algorithm

2.2 CONGESTION CONTROL IN DATA CENTERS

DC is a repository of computing and data storage resources linked together with several network devices, all working as one to run a number of services and applications with multiple needs and purposes. As their entire infrastructure is thoughtfully designed, they have the advantage of controlling everything that is connected to it, and they have hardware designed to achieve high throughput, low latency and maximum reliability on a range of network and

load conditions with minimum infrastructure cost and maintainability. Under the hood, the DC network has to support heterogeneous kinds of applications, like social media, data storage and many other kinds of services. It uses the high-bandwidth links, low propagation delay and small-sized buffers commodity switches to achieve its desired performance and allow these services to exist and coexist at the same time (AL-FARES; LOUKISSAS; VAHDAT, 2008; NOORMOHAMMADPOUR; RAGHAVENDRA, 2017).

However, due to the usage nature and complexity of a DC network, it pushes the boundaries, both physical and software-related, to its limits, and some unique issues may start appearing. One of them is the incast problem. Usually, the applications running in a DC have barrier-synchronized and many-to-one connection patterns, where multiple computers may simultaneously transmit lots of batches of data to a single aggregator node, which usually has a small buffer. On high loads of synchronized transmission on a single switch port, the network switch's buffers may overload, which will lead to packet loss and network performance degradation (WU et al., 2012a; ALIZADEH et al., 2010).

Another problem that may arise is the outcast problem. Data centers host several applications, all of them running together simultaneously, and for that to happen, they need to share resources. Resources like Central Processing Unit (CPU), memory and storage are evenly shared (depending only on the service settings), but network resources are not so easy to share evenly and, if unattended, let to compete against one another for their fair share of bandwidth. Ideally, the TCP should achieve true fairness, where the flows share an equal throughput in the connection. However, as the TCP was designed with the Internet in mind, its algorithms have a strong RTT bias, as performing the congestion follows strictly the state of the RTT, where low RTTs will receive a larger share and higher RTTs will receive less bandwidth. Yet, the design and topology of a DC network seem to alter this behavior. On a multi-rooted tree topology (which is a common network topology in data centers), whenever a connection is transmitting a large number of flows while another connection with fewer flows arrive at different input ports of the same switch, and are destined to the same output port, the connection with the small set of flows will have a significantly lower throughput than the larger one, out-casting them (PRAKASH et al., 2012). It is important to remember that this only happens on switches that use the simple taildrop queuing discipline, which is a common strategy used in low- to mid-end commodity switches, which are commonly found in DC networks.

2.2.1 Network-assisted control

Due to TCP's binary-based feedback architecture, it will only change its transmission state upon segment loss or duplicated ACKs (CHIU; JAIN, 1989). Whenever any loss is acknowledged in the sender, it will decrease the window value, and keep increasing while no losses occur. In turn, the congestion control works well enough in the originally proposed environment, like the Internet, it shows its weaknesses while performing in such a heterogeneous environment like data centers, where multiple kinds of applications are all competing against each other with

different characteristics. This is due to the way the regular congestion control algorithms gather their data. They only know what the sender and receiver knows, and cannot use any information from the network devices, like switches, to enhance their performance. At this point, algorithms that take advantage of the network devices come in.

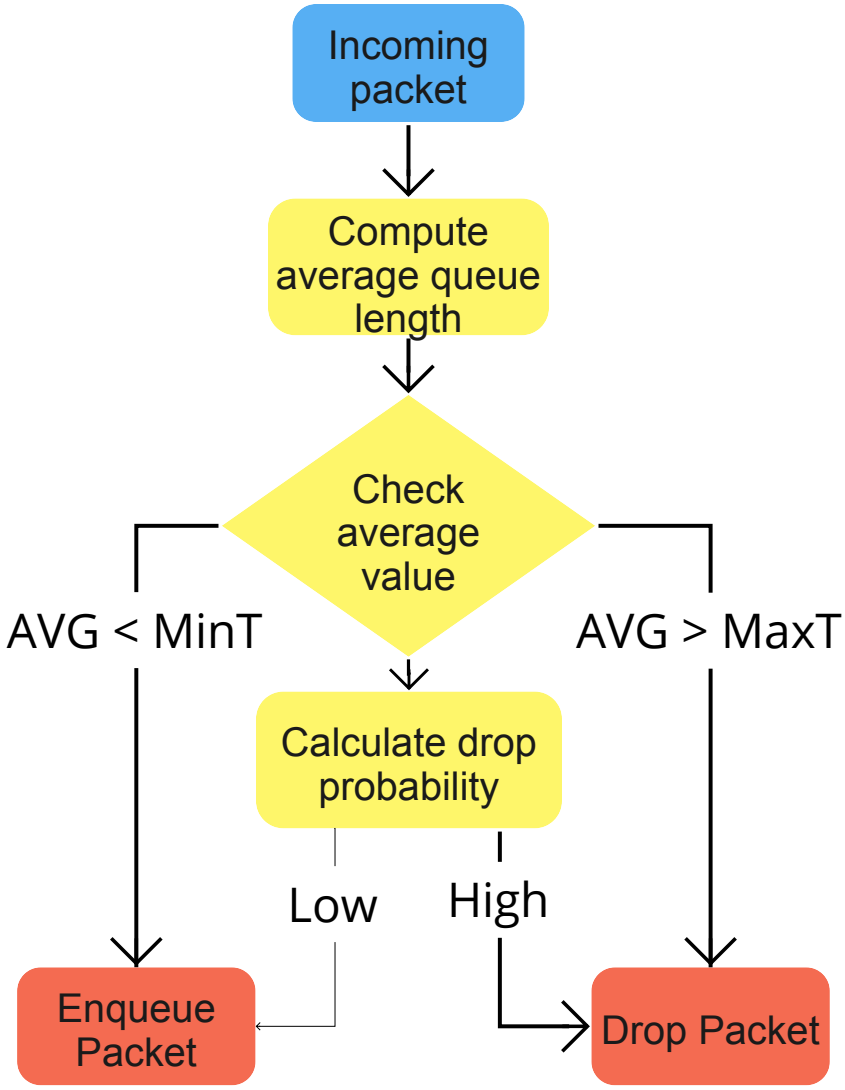
Explicit Congestion Notification (ECN) is an extension to TCP/IP stack on which it stops looking for packet losses and starts using information from the network itself (or the possible congestion notifications). When these notifications arrive, the sender may decrease the data output, just as it would if the CWND was full, and so, preventing the congestion from worsening. On the technical side, ECN uses 2 bits in the network layer. One for displaying that the connection supports the technology (ECN-Capable Transport (ECT) bit) and one for notifying that congestion occurred (ECN-Congestion Experienced (ECE) bit) (KUZMANOVIC et al., 2009). Under congestion, the network devices will populate the ECE bit in the network header. Then, the destination detects this bit and adds the ECE information in the ACK's header, in the transport layer. Finally, upon the sender receiving the ACK containing the ECN flag, it will scan for the ECE, and, if present, will decrease the congestion window in order to avoid or decrease congestion and then write 1 to the Congestion Window Reduced (CWR) bit, notifying that it applied the congestion control as expected (KUZMANOVIC et al., 2009).

The initial ECE writing must be done in the network devices, that support this kind of protocol. This requires constant monitoring in the forwarding queues on the device, followed by the trigger of certain actions based on the device's configuration. In this context, ECN works together with Random Early Detection (RED), a queue management algorithm for network schedule that uses a predictive model to preemptively drop packets that may lead to congestion (WU et al., 2012b). RED allows for the following parameters: link's bandwidth; minimum, maximum and instant queue size; average packet size; instant queue size threshold; and drop probability. RED also solves the TCP Outcast problem by replacing the taildrop packet strategy with itself, although it still does not achieves the true throughput fairness desired in a data center network (PRAKASH et al., 2012).

When the packet is received in the network, it goes to a series of steps to check if it goes to the queue or gets dropped. First, the RED protocol will check if the average queue length is less than the minimum threshold. If so, the packet goes to the queue and continues normally. If not, it checks if the average is greater than the maximum, and if so, it automatically drops the packet. If it is in between the minimum and maximum, it calculates the packet drop probability, and if it reaches a certain limit, it either gets queued or dropped (WU et al., 2012b). The Figure 3 illustrates the workflow of the RED algorithm in action.

Based on both technologies (RED and ECN) and to better attend data centers, a TCP variant was created called Data Center Transmission Control Protocol (DCTCP) (ALIZADEH et al., 2010). DCTCP aims to serve in a high-bandwidth and low-latency computer network. It incorporates the ECN protocol to notify the hosts of possible congestion, and has shown an improvement in lower latency, better buffer management and better data burst handling than

Figure 3 – Illustration of the workings of RED algorithm. Source: author.



the regular TCP (ALIZADEH et al., 2010). While the DCTCP does solve the incast problem in certain conditions (SURYAVANSHI, 2019), it fails to accomplish so when dealing with a large number of concurrent servers. DCTCP also has some issues of its own, as it may also start to mark packets very early, which may cause a too-much premature indication of congestion and cause an overall decrease in throughput. It also requires a significant amount of time for a new flow to obtain its fair share of bandwidth when there is another older running flow with a large window size (SURYAVANSHI, 2019). And as it makes a change in the TCP stack, it does require a change in the TCP server and client, and more importantly, in the network switches, which may be undesirable under a large network, which is a common case in a data center.

2.2.2 Virtualization of the congestion control

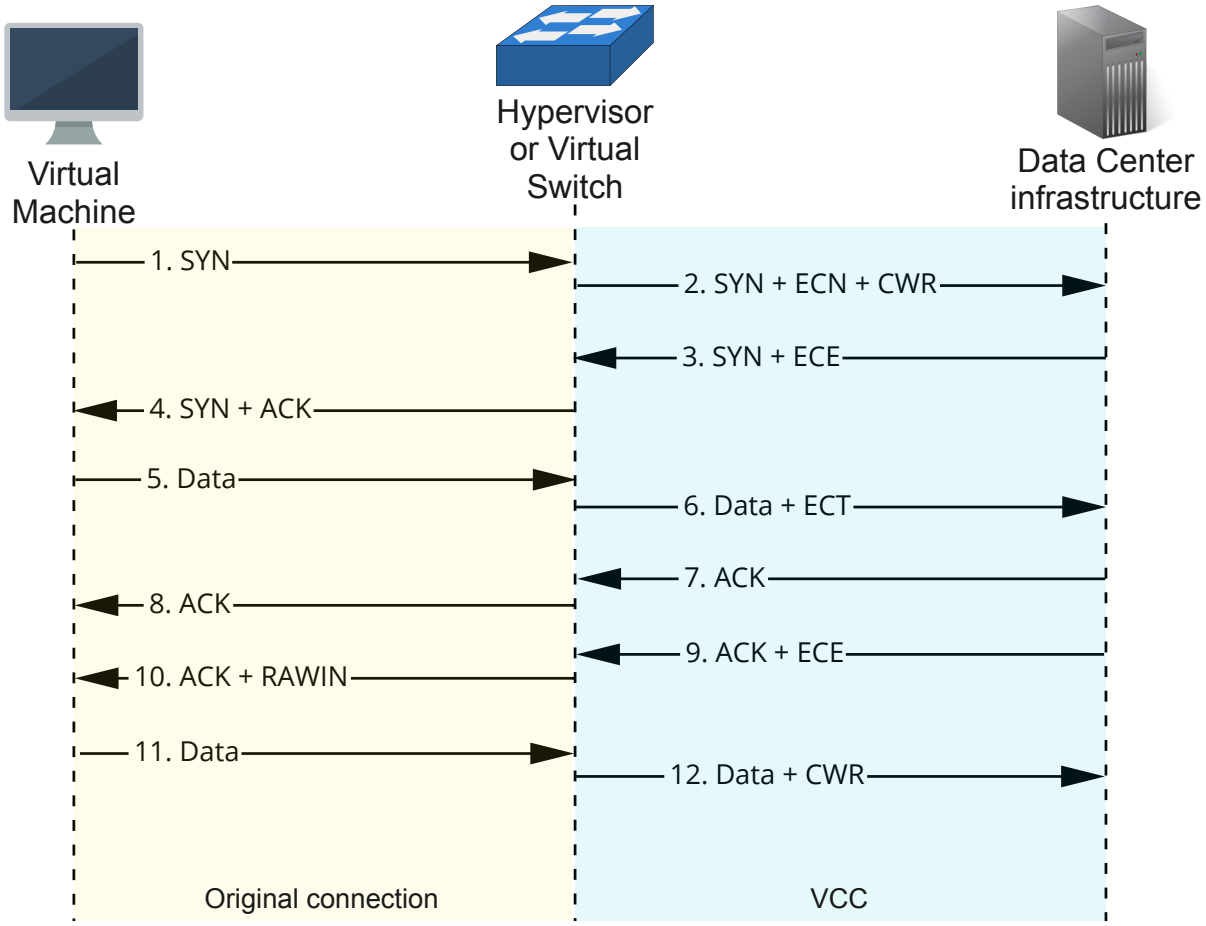
Data centers may have a wide array of different applications running inside them. Many of them provide Virtual Machine (VM)s to better use the machine's resources and facilitate the system's management. However, VMs may have different OSes running, meaning that each and every one of them may have a different congestion control algorithm for TCP at each time. This could result in one connection getting the most amount of resources than the others, which may be unfair and unwanted. The solution to this must deal with the fact that different versions of congestion control algorithms might be running, and it must not affect the behavior of the OS, or behavior of its applications and neither should it affect the system's integrity (HE et al., 2016; CRONKITE-RATCLIFF et al., 2016). These techniques that manage the congestion in VM environments are called Virtualized Congestion Control (VCC).

The VCC implementations are hypervisor- or switch-assisted, giving finer control over the network and its congestion control. VCC allows for different congestion control algorithms to be assigned to different internal connections, or flows, in the network (HE et al., 2016). The way the algorithm works within this environment that whatever the congestion control algorithm is used inside the virtualized OS, as there is no control over it, they interact directly with the hypervisor, which translates the algorithm data from the VM to the data center network, which may use ECN and RED internally, and then translates back (HE et al., 2016; CRONKITE-RATCLIFF et al., 2016; MORO et al., 2020).

The Figure 4 clarifies the behavior of the VCC algorithm. The left entity, named Virtual Machine, is any computing machine running on the DC. It's TCP calls are being forwarded to the VCC run time host (which can be either the Hypervisor or the Virtual Switch, represented as the middle entity on the figure), translating the congestion control algorithm back and forth from the VM to the DC (the right-most entity on the figure). This can be seen in the steps 1 through 4, where (1) is the original SYN packet from the VM, (2) is a "translated" SYN packet intercepted by the VCC, (3) is the original response from the DC and (4) is the "translated back" response intercepted from the VCC. As the connection goes on, more packets are being sent and translated by the hypervisor/virtual switch.

However, as the VCC solves an existing issue, it does have problems of its own. One of them is the need to translate the congestion control algorithms, which does have an impact on its performance. This impact is dependent on the load and the number of connections. When a large set of connections that are not optimized for the VCC algorithm are abundant, which is a common scenario, the VCC may deliver a slower performance than when not using it. As it does not communicate directly with the guest OS, the sender can only increase the sending rate as fast as the TCP implementation allows it (MORO et al., 2020; ISLAM; WELZL; GJESSING, 2019). It also is not a fully automated algorithm, as it requires a configuration beforehand, which is prone to human error as misconfigurations.

Figure 4 – Illustration of the VCC algorithm translating the TCP packets to and from the VM.
Source: Adapted from (CRONKITE-RATCLIFF et al., 2016).



2.3 SOFTWARE-DEFINED NETWORKING

Traditionally, computer networks have the data plane and control plane unified, on the network devices. The control plane contains protocol and algorithms that control the forwarding tables, which are special memory tables in every single network device (such as switches) that contain the rules for forwarding. The data plane is responsible for the data traffic, and is also bound by network devices. This plane follows the rules set by the control plane, and the same network devices (like switches and routers) have both data and control planes integrated. The data plane may also contain algorithms that optimize traffic in certain network conditions, like using devices from a certain manufacturer.

Having both planes on the same device was both a technology limitation and a desirable feature as it could potentially increases the forwarding performance, as the rules are right next to the forwarder (KREUTZ et al., 2014). As the computing resources have increased, and so has the desire to have more granular and deeper control over the network operations, the SDN paradigm has become more used in the DC network space, which brings to the table the idea of separating the data and control planes (KREUTZ et al., 2014; FOUNDATION, 2012). The

SDN paradigm has been in ascension for many years, and with it, a new way of handling and managing network connections has arrived.

2.3.1 Concepts and definitions

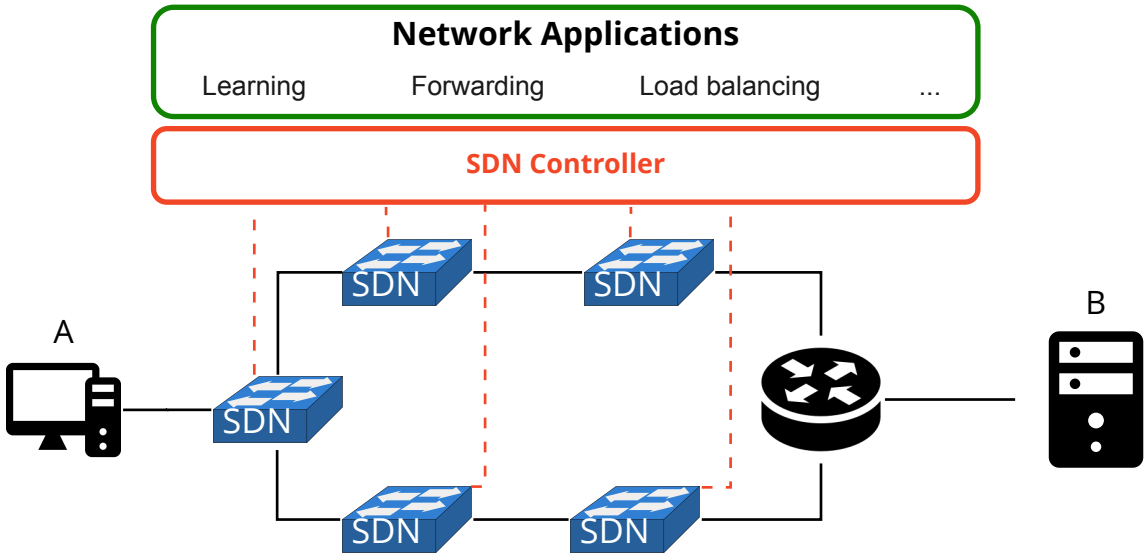
In a traditional network, each and every network device must determine its own forwarding rules, making use of some algorithms to help deciding them. On the other hand, SDN creates a new central entity called controller, which takes on the task of defining rules for the devices to follow, such as forwarding, dropping, and fall-backs. Network devices that support SDN will connect to the controller, and will start reporting their status to it while fully detaching the control plane from themselves. This ensures that the controller is the network's rule manager, making decisions based on all the network data available while still supporting devices that fail to support the protocol. Connecting every device to the controller also allows it to have a full view of the network topology and its content (KREUTZ et al., 2014; KIM; FEAMSTER, 2013).

As the name SDN suggests, this is a programmable environment and may change at any given moment, and that is due to the connection between the controller and the network devices, called the *southbound* connection. This connection allows the controller to send rules for the devices to follow, and that is how the controller manages the network (NUNES et al., 2014; KREUTZ et al., 2014). In addition, the controller also may expose an Application Programming Interface (API) that other applications may access and use as needed. This API may allow for configuring the network or just fetching data that is up to the network's administrator to decide. The connection that uses this API is called *northbound*. Finally, there is one last kind of connection a controller may have, which allows the controller to connect and communicate with other controllers in the same network, and it is called the *east-west* connection. This behavior is specially useful if the network administrator desires to have different controllers taking on different tasks (for example having one main controller for a more general forwarding while another controller for specialized rules, or a second controller as backup) (KREUTZ et al., 2014; FOUNDATION, 2012).

In Figure 5 it is shown how the SDN operates in a higher level. First a host in the network sends a packet to another host, or in this case, to the Internet. Then some switch that is connected to the controller receives the packet and must forward it to the correct port. As the switch does not have a defined table for that specific flow, it requests a rule from the controller. The controller then executes a series of algorithms pre-defined by the administrator to determine the rule of a said packet. Finally, the controller sends the rules back to the switch which will then execute the determined action, and depending on the rule, the controller may apply it to other switches. The controller itself and the link between the controller and the switches are in red, representing the southbound API. The switches, the blue boxes in the middle of the network, represent network infrastructure. And last, the network applications box in green represents the northbound API (FOUNDATION, 2012).

SDN makes use of tables on switches in order to connect all of the networks layers. The

Figure 5 – Illustration of a generic SDN. Source: Adapted from (KREUTZ et al., 2014).



process is simple: the controller manages the tables for all devices while network devices use their own local table to fetch and apply network rules. There are many kinds of tables in a SDN environment, and that depends on the protocol that implements it, but the most common ones are the flow, group and meter tables. A common behavior between all of the tables is the way the entries are searched by SDN. Each entry is composed of a key-value pair, where value is the rule or outcome and key is the identifier. Although it depends on the protocol in use, the key is usually a tuple specifying the entry, using attributes such as network addresses, ports, priority, target and many others.

In contrast to traditional networks, which define the path of a packet only by the source-destination pair, SDN networks use flows to determine their trajectory. Flows are stored in a specific memory table, called the flow table. The flow table is available for each network device, and each entry on this table defines a tuple of information (which varies depending on the settings of the controller) that characterizes each connection to a flow. The information in the tuple can be as simple as source, destination and type of packet to more complex data like switch exit port or packet size (KREUTZ et al., 2014).

Anytime a packet is processed in the device, it tries to match that packet's properties with any entry in the flow table. If it matches any entry, it fetches the set of actions defined in that entry and applies to that packet. This set of actions may be to forward to some port, apply a transformation to the packet itself (like a header value), or even drop the packet entirely. If no matches occur, the device will first send a request to the controller informing of this unknown packet and then proceed to follow a default action or, if none are present, do nothing (KREUTZ et al., 2014; FOUNDATION, 2012).

Another possible action is to redirect the rule to any entry in the group table (KREUTZ

et al., 2014; FOUNDATION, 2012). The entries in this table allows for an easy way of grouping certain common rules to be applied on certain packets that follow a common characteristic, which may be desirable depending on the situation. One last table is the meter table, which contains per-flow counters and enables Quality of Service (QoS) operations to be applied to that specific flow, such as constraining the flow into a certain bandwidth. When the controller is setting a rule, it is actually writing to the desired switches' table via the southbound connection.

On the note of SDN protocols, OpenFlow (FOUNDATION, 2012) is a well-known protocol used extensively in both academic and professional areas. OpenFlow has had many versions throughout its development, ranging from 1.0 to 1.6, but the most well known and supported version is 1.3, which is why every reference to OpenFlow of this thesis is based on OpenFlow version 1.3.

2.3.2 Congestion control in a SDN environment

While SDN itself is not a technology that acts in any way as a congestion control algorithm, it is a paradigm that enables network developers to do so. Given that the controller has the knowledge of the entire network topology, it allows for the creation of forwarding algorithms that calculate the best possible route a priority packet may require and installs that rule in every switch in the network. In addition, using the meter table, it is possible to allow a certain flow to have less bandwidth available while another one, that has a higher priority, has the full path available (DIEL; MARCONDES; KOSLOVSKI, 2017; DIEL; KOSLOVSKI, 2019). Furthermore, as the controller can inject rules into SDN devices, it may as well inject a rule to intercept ECN packets and extract their value, and thus, allowing for one more information source.

It is worthwhile to mention that these are only ideas, but the capabilities SDN provides make for endless possibilities when designing a congestion control tool. The flow table also give the network administrator to apply rules on very specific connections that may be impacting the rest of the network, by using a more comple tuple to represent the flow entry. This enables the algorithm to apply actions such as lower the bandwidth or modify the packet's header information in a way that helps the overall network performance.

2.4 CONSIDERATIONS

This chapter has presented a set of information surrounding the field of congestion control in a TCP connection with data centers in mind, which serve as a base for this work's proposal. The current scenario of TCP congestion control can be separated into two categories: end-to-end algorithms and algorithms backed by network devices. The end-to-end algorithms like CUBIC, Reno and New Reno have been upgraded from a base algorithm that is focused on a regular user's use case, such as Internet browsing. Algorithms that use information from network devices, which are most commonly located in DC, have the advantage of having the network

itself contribute to the operation of the said algorithm, and thus, may deliver better performance under said architecture.

However, both methods have their advantages and disadvantages. End-to-end algorithms lack the information of the network state, and may perform poorly under changes in the packet's path, and may even start a congestion workflow when not needed. It also cannot handle the high-bandwidth, and low-latency DC require. On the other hand, network-aided congestion control, such as ECN, are an alternative to the default TCP stack algorithms, but they are not yet widely adopted.

Meanwhile, a new network paradigm called SDN has reached a wide range of DC around the globe, with the addition of controllers that facilitate the network's management. Along with controllers, SDN also brought the concept of flows, which are individual connections inside the network that have certain parameters, like the same port, the same end-to-end nodes, the same packet types, and so on.

The SDN controllers have tools that allow network developers to write software that connects with it, and may feed data from the network into an algorithm. This algorithm may return back and request to change the network's behavior in some way that benefits its needs, like a congestion control algorithm. Being a programmable interface, it is easy for developers to write more and more complicated software that runs in a SDN environment than it is to write in a traditional network, and that is the basis for using SDN in this proposal.

3 MACHINE LEARNING APPLIED IN CONGESTION CONTROL

This chapter brings into light a new concept required for the full understanding of the present work proposal, termed Machine Learning (ML). In short, ML is based on mathematical models that are designed to train algorithms capable of performing specific tasks, such as pattern matching, classification and predictions.

With that in mind, this chapter brings a brief summary of ML in Section 3.1 along with the most important concepts of this subject and how it attaches in network congestion control. This section also specifically presents the AC method, which is a Temporal Difference Learning (TDL) based method for the unsupervised learning algorithm, being the chosen method of this proposal, and therefore, it is required for the understanding of this work. In Section 3.2 it is discussed how the data flows in a network-focused ML environment and how such technology can be handled in this proposal. Section 3.3 presents some ideas that could lead to full solutions using ML algorithms and techniques that could positively impact congestion control on networks. Finally, in Section 3.4 it is presented the partial considerations for this chapter and what we have learned with ML and how it could be used to solve this problem.

3.1 CONCEPTS AND DEFINITIONS

ML uses mathematical models to enable a way of scrutinizing data and generating knowledge. Not only does it creates static knowledge, but it is also able to synthesize new experiences and improve over time, in an ever-lasting self-iterating training set (GENTLEMAN; CAREY, 2008; BOUTABA et al., 2018). The basic essence of ML is to synthesize a large data set, identifying and matching hidden patterns in it. These patterns can then be used to generate other data, such as groups or identifications for existing elements.

Recently, the state-of-the-art ML algorithms have become more and more flexible with faster learning, allowing resilient techniques to be applied in real-world scenarios, breaking through the previous known limit of traditional programming paradigm, and allowing the development of thought-to-be impossible applications (BOUTABA et al., 2018), with some good examples being search engines and face detection. Indeed what makes the ML the success it is, is not the composing algorithms, but the data used to train them. While a good algorithm might make it faster to learn or more difficult to reach false positives or other problems, the real deal when dealing with ML is meaningful data, and that is absurdly abundant in today's networks, which are bound to grow even further.

In the ML universe, there are four major categories of problems:

- **Clustering:** The objective of this problem is to group similar data and increase the gap on known groups;
- **Classification:** The objective is to categorize data in an ever-increasing set of discrete values (or labels);

- **Regression:** This problem requires the application to map a set of inputs to a set of continuous values or a function to allow the prediction of future values; and
- **Rule Extraction:** Requires the application to identify any statistical relationships in the data set and create high-level rules (or associations) out of it.

These four problems are the basis for machine learning algorithms in the current days. Along with these problems, there are also four major paradigms for the algorithms: *supervised, unsupervised, semi-supervised* and *reinforcement learning*. Each of which has its own methods of collecting data and the technical design application (BOUTABA et al., 2018; ERICKSON et al., 2017). The following subsections describe the four most common machine learning methods: supervised learning, semi-supervised learning, unsupervised learning, and reinforcement learning.

3.1.1 Supervised Learning

Analogous to "a teacher teaching its student", this technique uses labeled training datasets to generate the mathematical models, meaning that the data must previously have been labeled, or identified. The data set (composed of n elements) usually comes in pairs, each containing an instance x and a label y: $\{(x_i, y_i)\}_{i=1}^{n}$, and when it comes in this (instance, label) pair, it is called *labeled data*. Typically, this approach is used in *classification* and *regression* problems (ZHU; GOLDBERG, 2009; BOUTABA et al., 2018), and is used to learn to identify the patterns hidden in the (instance, label) pairs, where a new instance of the problem will receive a matching label.

3.1.2 Semi-Supervised Learning

It is possible that only part of the data has been correctly labeled, called partial knowledge, and that could manifest as incomplete or missing labels. In these cases, the semi-supervised learning is employed, and as its name suggests, it sits between supervised and unsupervised learning, so much that many semi-supervised learning algorithms are extensions of either method to include additional information (ZHU; GOLDBERG, 2009; BOUTABA et al., 2018). The data elements (n) comes in both labeled ($\{(x_i, y_i)\}_{i=1}^{n}$) and unlabeled ($\{x_j\}_{j=l+1}^{l+u}$) instances, being u the unlabeled instances and l independent identical examples, and the goal of the semi-supervised learning is to achieve a better performance than its supervised counterpart with only the labeled data.

3.1.3 Unsupervised Learning

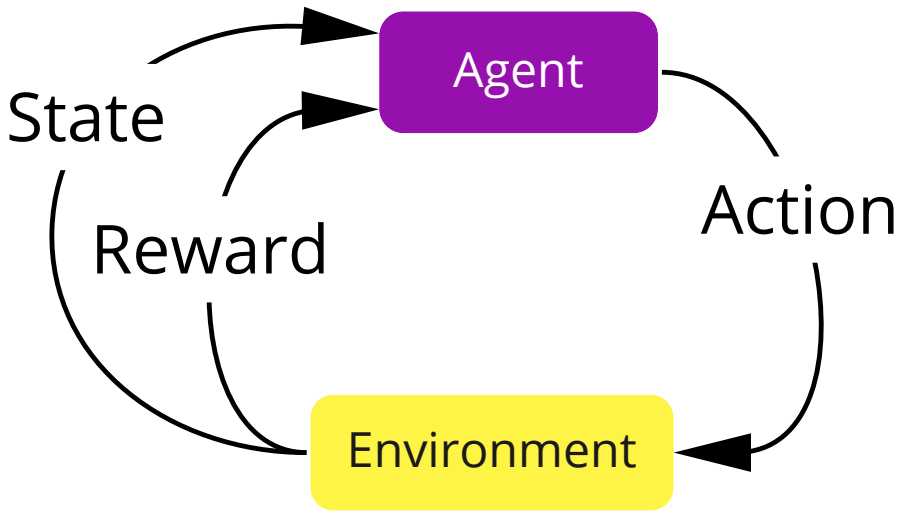
On the other hand, unsupervised learning uses totally unlabeled data instances $\{x_i\}_{i=1}^{n}$, and uses it to generate models that are able to search and discriminate hidden patterns in the dataset (GENTLEMAN; CAREY, 2008; ZHU; GOLDBERG, 2009; BOUTABA et al., 2018). The most common problem addressed with unsupervised learning is the clustering problem, such

as detecting outliers and density estimation, which both can be used in a network context, such as security and flow control.

3.1.4 Reinforcement Learning

This method uses an agent-based iterative algorithm, which keeps on modifying its mathematical model to achieve better performance and decision making. The training data of RL is a set of state-action pairs and rewards, which can be positive or negative. The agent of the model iterates on itself based on the feedback of its actions on the environment, and learns how to perform better, as well as the best sequence of actions for a certain state on each iteration. As the origin for RL is the Markov Decision Process (MDP), it shares some parameters with it, like the environment and its states. The environment is the set of possible agent states, usually noted as S, and in each state, there are actions A on which the agent can take to go to another state. The Figure 6 illustrates how the algorithm works. The Agent (represented as the purple rectangle) performs some action on the environment (yellow rectangle). Then, the new state of the environment is collected and sent to the agent to perform some new action on it. At the same time, using the data of the old and new states, a reward is calculated and sent to the agent so that it can learn.

Figure 6 – Illustration of the flow of a reinforcement learning algorithm. Source: author.



The RL uses a *policy function* to map the states to actions: $\pi(s) : S \rightarrow A$. The value function is another component of the RL method and represents how desirable for the agent it is to be in that certain state, and it is equal to the expected reward for an agent in the state s. This function usually depends on the policy by which the agent chooses the action to take.

In short, the goal of the RL algorithm is to find the optimal policy in which it specifies the correct action to take in each possible state, maximizing the reward. This mechanism could

be employed for a rule extraction problem, where it is needed to develop patterns and generate a statistical analysis of the data while keep on learning and adapting.

Differently from the other three methods (supervised, semi-supervised and unsupervised), RL does not have an immediate optimal performance, but instead, keeps on enhancing in as the algorithm keeps running (SUTTON et al., 2000; TESAURO, 2007; BOUTABA et al., 2018; FADLULLAH et al., 2017). This behavior makes it even better in scenarios for decision-making, planning and scheduling.

3.1.4.1 Temporal Difference Learning

Inside the subset of reinforcement learning techniques, there is one in specific called TDL. TDL is a combination of Monte Carlo and dynamic programming methods and ideas, that learn from raw experience without a model of the environment. This method bootstraps from the current estimate of the value function, which means that the algorithm updates the estimates based on part on other estimates, which have already been learned, and keeps adjusting these parameters gaining more and more accuracy before the final outcome becomes available. This idea is based on the way animals learn (SUTTON, 1988; SCHULTZ; DAYAN; MONTAGUE, 1997).

TDL is also considered model-free, which means that it is an algorithm that does not use the transition probability distribution associated with MDP. Instead, TDL uses the transition probability distribution and the reward function as the model of the environment, which is why it's called model-free (TESAURO et al., 1995; TESAURO, 1992).

With the difficulty involved with congestion control, and the vast amount of data and possible configurations of a network, the RL, specifically the TDL, has the most advantages among the other methods of ML. The continuous evolution of the model throughout its lifespan is a desirable key point to have in a network environment, and the problem it is most useful in being the rule extraction makes it the most obvious method to use in this proposal (TESAURO et al., 1995; TESAURO, 1992). The temporal difference learning method also allows for better accuracy in the training process, as it does not require the final outcome to be available outright during training, but only at the very end, which in a network environment is a common case as the network does not have an immediate stop point.

3.1.4.2 Actor-Critic model

In the vast universe of reinforcement learning, one particular method called AC draws attention. The AC method is a model-free TDL method that employs two distinct structures in-memory in which it represents the policy table independently of the value function. In the AC method, the policy structure is known as the *actor*, as it is the one to take actions, while the value function is the *critic*, as it criticizes the action taken by the actor, which comes out as the reward of each action. In this method, the critic must learn how to criticize to latter critique the actions

the policy follows, in the form of the temporal difference error (as discussed in Section 3.1.4.1). This is the critic's output, and is used for both critic and actor to learn (KONDA; TSITSIKLIS, 2000; BHATNAGAR et al., 2009).

In summary, the temporal difference error comes in the form of:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

in which V is the value function employed at the critic, s is the state and r is the reward, both at the current state (t), and for the next state ($t+1$). This resulting error can then be used to evaluate the selected action a_t in the state s_t. If the error is positive, the probability of selecting a_t is greater, and if negative, the tendency decreases. In addition, being $p(s,a)$ the value at a time t of the policy parameters (indicating the probability of selecting each action a when in each state s), the increase or decrease of the probability is given by:

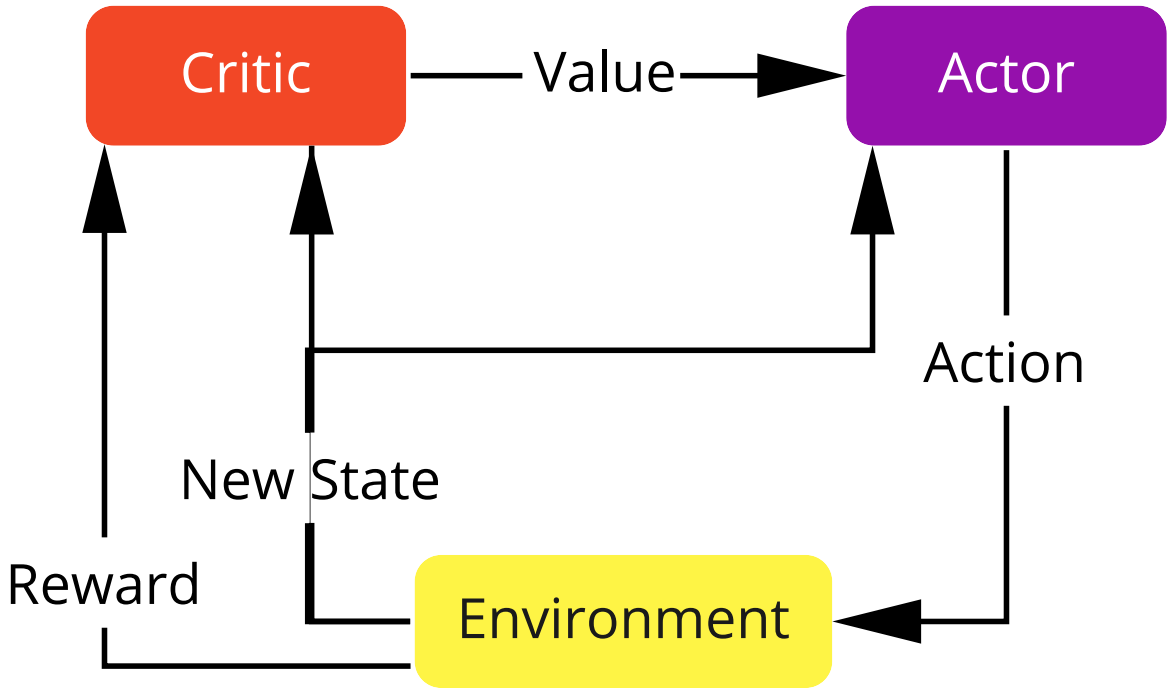
$$p(s_t,a_t) \leftarrow p(s_t,a_t) + \beta\,\delta_t$$

in which β is a positive floating point parameter defining probability.

The Figure 7 illustrates how the Actor-Critic method works. The actor performs a chosen action, which affects the environment. The environment will generate a new state and a initial reward value with predetermined deterministic rules. Those generated values will be sent to the critic, which will compute a new reward value and forward it to the actor. The actor will then use this new reward value to learn. The critic will also learn by comparing the reward it received, the reward it generated and checking how the new state has changed from the prior.

The AC method has two main advantages over other TDL or even RL methods. The first one is that the cost to take action is minimal, requiring little to no computational effort. This could be a significant improvement in scenarios where the action set is actually a continuous value, as there is a need to iterate over a possibly infinite action set as other methods. The second advantage is that it can learn the optimal probabilities of selecting each various actions, which can be advantageous in certain scenarios (JAAKKOLA; JORDAN; SINGH, 1994; BHATNAGAR et al., 2009). These advantages have given the AC method the ability to learn in big and complex environments.

Figure 7 – Illustration of the flow of a Actor-Critic algorithm. Source: author.



3.2 DATAFLOW OF ML-BASED CONGESTION CONTROL

The core component of any machine learning project is the dataset. With a good dataset, a simple algorithm can generate a perfectly good model, while a poor dataset may, even with a highly advanced algorithm, generate a biased under-fitted model. The same principle is applied to the computer network environment, even so when dealing with congestion control (BOUTABA et al., 2018).

The first important step in modeling a ML algorithm for the network is the data collection. In a network environment, the data collection can be both *online* or *offline*. Offline data is when using pre-collected archived dataset, which can then be used for training and testing the model. Some big entities, like Facebook (FACEBOOK, 2017), Google (Measurement Lab, 2009) and the University of Waikato (GROUP, 2013) do share their archive network dataset, with various amount of complexity, detail and size. However, the data in such a dataset is static and will not reflect actions and decisions taken by the model. For this purpose, online data collected in real-time, as the environment is up and running, may serve better. In general, online data can be used for training, testing and feedback to the model, or even re-training when needed.

One way of collecting online data is by using measurement and monitoring tools, which gives control over the collecting rate, what to collect and how to save. Monitoring in the network can be either active or passive. When passively monitoring, the tool will try to fetch as much

data as the host machine has, like the amount of packets it is receiving, the latency of each packet and the types of packets in the traffic. These kinds of tools only consume the host's resources, and have zero impact on the network, but they do sometimes require a greater number of running instances of the tool, typically in different network nodes to enlarge its measurement coverage. On the other hand, active monitoring requires the injection of packets designed for measurement. These packets may measure how the network behaves, and detect properties while traveling through the topology. The active tools must be used with caution, as they do cost an additional overhead over the network, which may have its bandwidth reduced and see an increase in latency (FRALEIGH et al., 2001).

Along with the network tools, that do require external programs to run, it is possible to use SDN's capabilities and measure directly from the controller, which can be done by accessing the northbound API, usually through a HTTP server. When using SDN for data collection, the data available varies from protocol to protocol. For example, OpenFlow, which is one of the most used SDN protocols, allows data from both the flows and switches to be collected, which include a vast range of properties like trafficked packet count, trafficked bytes count, bandwidth of each flow, the rules applied and many more (FOUNDATION, 2012).

Once the data is collected, sometimes it has to go through a process of cleaning, in which the collected raw data is cleaned into something usable, and the unnecessary or unrelated data is removed. This step is important depending on a set of parameters, like the kind of problem, the kind of ML method used and the final result expected, but unwanted data may lead to over- or under-fitting and bias.

The moment the entire data is ready to be used, it is first split into three subsets, the training, testing and validation sets. The training set is the one used to train the mathematical model, in which the algorithm will iterate over and over trying to generate the ideal parameters for the model, which can depend on the algorithm used. The validation set is then used if there are multiple possible models to select, which some algorithms may generate. These different models have different parameters, and the validation phase will pick the one that performs best. This validation step is not necessary when there is only one model, which can happen in some algorithms. Once the model is selected, then the test phase starts. This uses the testing set, and it has the goal of estimating the unbiased performance of the model.

The training set is usually the largest one, as it is the one that will actually have an impact on the model, and the configuration will usually be 60% for the training set, 20% for the validation set and 20% for the testing set, but when no validation is required, this ration is commonly 70%/30% or 80%20%. But in extreme big data environments, this ratio may change to extreme compositions like 98%/1%/1% or 99%/1% (BOUTABA et al., 2018).

Once these steps are done, the model is ready to use. After this stage, there are some meta-information that can be extracted from the model, such as the true labels and classes. This usually requires some manual labeling, and does require knowledge in the field, but there are some automated techniques. This step is crucial to determine what the model actually learned.

For example, if classifying traffic data, the labels should be something that makes sense for the goals, like the kind of packets trafficking, or the size of each flow. This meta-information is called the ground truth, and if it is possible to successfully extract it from the model, then the model is ready for real-life usage.

3.3 CONGESTION CONTROL OPPORTUNITIES

In spite of everything congestion control requires to be solved, ML can be of great use in this area. There are a couple of approaches that could be taken when uniting ML algorithms with congestion control techniques. Of them is by using classification algorithms. Their purpose is to return a single value that represents a major group of a given input. These kinds of algorithms could be used for scenarios where the network admin has some knowledge of congestion resolution and knows some possible actions, but cannot know when to apply each. Classification algorithms could help identify these cases and applying correct rules. Here are a couple of ideas using these techniques:

- Select the optimal congestion control algorithm by feeding some of the network information into a classification algorithm, which outputs which algorithm to use. This idea takes advantage of the fact that ML classification algorithms are extremely fast to output a result, and that it is possible to change the congestion control algorithm on a host PC;
- Another idea is a classification algorithm that could help on the congestion control problem by defining the type of traffic inside the network. With this definition, a set of rules could be applied throughout the network (or send those rules directly to applications running on the network to adjust their traffic accordingly). As an example, if the algorithm classifies the network as undergoing bursts of requests, then it could scale up the application horizontally;
- Finally, it could be interesting to use a classification algorithm to identify strange or bad behaviors on the network and take actions blocking those from occurring. This could be aligned with security concerns and create a product that stops not only bad traffic from overruling the network capacities (like Denial of Service (DoS) or natural network congestion) but also avoids unwanted traffic from spying on the network.

Another popular ML technique is regression, which takes an input and, based on historical data, outputs a prediction related to that value. Regression is most useful when the implementation of a deterministic algorithm is considered hard or slow, or even impossible with current knowledge. Following are some ideas using the regression technique:

- One idea is that for each computing host on the network, deploy a new congestion control algorithm where its buffer sizes are controlled by a ML algorithm. This idea makes use

of the algorithm's capability of generating predictions based on past experiences, which means that an algorithm that behaves somewhat similarly to Slow Start might have fewer drops and even higher increases, optimizing the network usage;

- Another approach would be to implement a forwarding or routing algorithm using the regression technique, which chooses the optimal route given all the possible routes available on the network. This could reduce the congestion by having a more efficient infrastructure usage;
- Finally, another idea would be to have a behavior prediction algorithm, which could predict future changes in network traffic and auto-adjust some settings, like QoS, routing or event some other network rules.

It is important to keep in mind that ML algorithms will not always return the optimal solution. Depending on the scenario, type of calculation, the quality of training data and other factors, these kinds of algorithms could indeed return the optimal solution, but that must not be expected from them. Usually, the outcome approximates the optimal solution, sometimes they are equal to the optimal solution, and sometimes, however, they have an erratic value and return a seemingly nonsense result. All of these behaviors must be taken into account when developing with such types of algorithms and techniques, especially when working on the training of these algorithms.

3.4 CONSIDERATIONS

This chapter has presented the basis of ML, and the main problems it tries to solve, such as clustering, classification, rule extraction and regression. It has also presented the four main methods of ML techniques, supervised, semi-supervised, unsupervised and RL. Not only that, but it also gave some information on the reason this proposal adopts the usage of the RL, being that its iterative learning process keeps the mathematical model up-to-date with the environment, which is extremely necessary in an environment such as the computer network, which keeps on changing indefinitely.

Specifically the AC method was reviewed, which is a RL method used in some scenarios. It brings concepts from both value-based and policy-based RL methods, combining them in a best-of-both-worlds algorithm. Employing the actor as the policy gradient update and the critic as the validator of each action brings greater efficiency to the Temporal Difference updates each step. The benefits of the AC method draw attention to itself, and thus, this proposal adopts its usage.

This chapter has also brought how to combine ML with a problem in the computer network environment. What tools to use, how to fetch data for training and what to watch out for. It also has shown how SDN can be used to harvest data from the network using the northbound API, and access data not previously accessible. This allows for more robust solutions that need

little to no changes in the network, only the SDN itself. And bringing to the specific issue of this proposal, computer network congestion control is a hard problem, containing a vast amount of variables that can change in any second. This means that if an algorithm were to oversee an entire network's congestion control management, it would need to be extremely flexible and learn how to operate in innumerous scenarios. These requirements point to a RL as the ideal solution, and the AC method takes an extra step as it has advantages in large scenarios (like in a computer network), which is why this proposal employs its usage.

Finally, some ideas have been pointed out for some of the most common ML techniques, classification and regression, which could lead to applications and techniques that help to manage congestion on networks. Although some ideas are more complex than others, they do show that using ML for congestion control is not only possible but a good combination. ML algorithms have the power to process huge amounts of data in a reasonable time that the network application requires and also powerful behavior and value prediction tools which could be extremely advantageous in certain situations where the network could proactively change some parameters in order to avoid congestion, or responsively alter them to control ongoing congestion.

4 REINFORCEMENT LEARNING- AND SDN-AIDED CONGESTION AVOIDANCE TOOL (RSCAT)

This chapter brings the proposal and description of the RSCAT tool. For such, this chapter has been divided into 4 sections. In Section 4.1 it is presented how the current state of congestion control algorithms are handling real-world scenarios, such as TCP-based networks and data centers. It also presents how the discussed technologies such as SDN and ML are used or may be used to help control network congestion. Section 4.2 presents the main proposal: RSCAT, a SDN- and ML-assisted tool for congestion avoidance. Finally, Section 4.3 presents the related work, while Section 4.4 discusses the final remarks for this chapter.

4.1 CURRENT CONGESTION CONTROL STRATEGIES

Considering that the Chapter 2 brings on the problem of computer network congestion control, it is easy to perceive that solving such a problem is a complex and challenging task. There are many algorithms and methods for reducing the impact of network congestion, and even good algorithms that may control it to some degree. Most of them depend on the TCP protocol, such as Reno, New Reno, Vegas and others, each bringing in new ideas, implementations and sometimes some new problems. It is important to keep in mind that, while large-scale traffic may be the root cause of a network congestion, it is not the only culprit.

The TCP's implementation of retransmission algorithms is what makes it one of the most used transport-layer network protocols, as it guarantees the order and the receivement. When it detects that a sent packet may not have been processed or even received at the destination, it re-sends that packet until it exhausts its tries or until the packet is properly received and acknowledged. This retransmission may take part in TCP-based network congestion, as it triggers whenever the latency between the sending and the acknowledging increases, and when the network is saturated with other connections, this creates a positive feedback loop, in which the TCP thinks the packets have been lost and retransmits them, causing more packets to be around the network, causing the latency to increase feeding this cycle.

Many congestion control algorithms try to optimize the congestion detection and when to retransmit the packet, but they are limited to the data available on both ends of the connection, meaning that whatever is going on in the middle of the network is ignored. The sole fact that the data used is only the data available at the ends of the TCP connection already makes this problem hard to solve.

In the case of Data Centers which also use TCP-based connections, there are some congestion algorithms that take advantage of the control available over the entire network, and use the network devices in the middle of the network to help avoid congestion. ECN is a good example of such an algorithm. It relies on capable network switches and uses data inside the header of network packets to signal congestion state throughout the network. But these kind of algorithms do present flaws, as they do require the support of network devices (which is not

common) and do not perform as well as they could, prematurely signaling congestion more often than not.

Network-assisted congestion control has shown that there are advantages of using the knowledge in the network itself, and in this sense, Section 2.3 presented the SDN protocol, in which the controller keeps on monitoring the entire network, gathering information about traffic, transmitted packets, topology and much more. It also brings new controlling capabilities to network administrators, such as a system of setting up traffic engineering rules, like forwarding, QoS tools and even changing TCP's header information. SDN has shown to be of great use in DC environments due to its expanded arsenal of administrative tools (KREUTZ et al., 2014). By combining the tools that SDN provides with the historical knowledge of congestion control algorithms it is possible to see that the algorithms could benefit from the knowledge stored inside the SDN controller. Adding to the fact that the congestion control algorithms use TCP header to store information across both ends, it is possible to use SDN's capabilities to change in real-time and as-needed the congestion control data stored inside TCP packet's header for a better algorithm performance.

Due to the fact that the SDN controller has full visualization of the network, it could be used to figure out which connections are causing the network congestion, and what could possibly be the most optimal value for the RWND (or CWND) for each connection. Specifically regarding the RWND values, SDN not only can it be used to calculate the ideal values, but it can also be used to directly modify the TCP packets, allowing the optimization of already existing TCP congestion control algorithms without changing anything on the server, host or the network, as this new algorithm could be a simple SDN application running alongside the controller.

But even though the SDN brings the tools necessary to perform such an idea, it is still hard to define what are the optimal values. When dealing with networks, it is possible to be dealing with thousands of nodes in the network, with hundreds of thousands or even more simultaneous connections. This means that the data gathered in the controller is huge in size, and traditional algorithms might have a hard time processing such a vast amount of data. There is also the problem of building an algorithm that is both efficient, flexible and adaptable. It needs to be efficient so that its decision-making does not take too much time to calculate and ends up being too late to act. It must be flexible as the connection are not stable at all, as new connections are being created and destroyed all the time. And it must be adaptable as new devices can be connected or removed anytime, so fixed rules are not an ideal way to go about it. So that is when the ML algorithms come in.

ML algorithms are model-based algorithms, meaning that they work around an mathematical model that acts like an intelligence. This mathematical model is flexible and as iterations keep happening, it keeps synthesizing the outcomes and thus, learning. This process means that the learning phase takes a long time to reach an optimal intelligence. But when it does reach, it starts to make decisions by only looking up into a mathematical model trained table, which is computationally fast, and, if trained properly, will return near-optimal results for the

desired scenario. This means that ML algorithms are suited for large data workflows, and can help scenarios where speed is key and the data amount is large.

4.2 PROPOSAL

Summing everything up brings the proposal of this thesis, the RSCAT, a ML-driven and TCP-based congestion control mechanism that uses SDN features to track down information and to figure out the optimal congestion windows for each network flow. Specifically, ML is applied at two different stages running atop the controller: initially, a pre-trained classification algorithm is used for detecting network congestion, while when it comes to deciding which value each connection should have for the congestion window, a more robust AC algorithm will be used. The AC method has been proven to be a good candidate in possible infinite-state scenarios (as discussed in Section 3.1.4.2), like a network is, and it also can respond quickly to value requests, which is a key necessity for such a tool as the faster it changes the congestion windows, the faster the congestion will decrease.

4.2.1 Requirements and definitions

With the purpose of solving a subset of the problems stated in Section 2.1, this chapter brings a method of controlling congestion control with the aid of ML capabilities and SDN functionalities. For the proper development, control and testing of RSCAT, requirements, both functional and non-functional need to be set and followed. The environment requirements, which are the minimal configurations needed for RSCAT to run in a certain environment, are as follows:

- The network must contain switches supporting the SDN protocol OpenFlow version 1.3 or higher; and
- The controller machine must have the necessary hardware configuration to run RSCAT.

The functional requirements are the actions that the RSCAT tool needs to execute properly in order to be considered finished and functional, and they are:

- RSCAT must detect network congestion within the SDN scope;
- RSCAT must generate SDN network rules which, when applied, leaves the overall performance on the network the same or better than it was before applying;
- RSCAT must not negatively impact the overall network performance;
- RSCAT must start executing immediately with no manual inputs required; and
- RSCAT must connect with the existing SDN controller and start monitoring the network, classifying the congestion and running all of its modules when required.

4.2.2 TCP avoidance technique

Independently of the TCP algorithm used by the DC network, there are multiple parameters available to examine and control the final performance, being them from the algorithm itself or the network parameters. In this work we are interested in parameters that do not require an update on legacy hosts, operating systems, and modules so that no changes are required by those systems. In this sense, as the RWND management in TCP is currently implemented, it can be temporarily used to limit the CWND value of each connection, as a means of congestion control and avoidance (CRONKITE-RATCLIFF et al., 2016; HE et al., 2016). This means that it is possible for the RSCAT to manage, even so temporarily, the congestion control mechanism employed in connections of the network to help the network without hosts' modifications. In other words, RSCAT can be implemented at a SDN controller level and work together with multiple TCP variants transparently, meaning that the network is unaware of its presence. If eventually required, in cases where a certain persistence is needed for some tasks to perform correctly, RSCAT will have to inject some TCP segments to reconfigure the ongoing scenario.

It is worth mentioning that while RWND can be updated for each TCP segment, the buffer configuration is a OS-based information. In other words, the first approach can be transparently implemented at controller level, however, it is required to intercept, update and re-inject some segments to reconfigure the scenario. In turn, the buffer configuration requires an intrusive update and consequently an administrative control (common in DC scenarios). As the buffer configuration is required for a stronger, less-invasive and lasting solution, it is not required for this proposal, and can be seen as an extension to it, and, if desired, it could be implemented in an on-demand way.

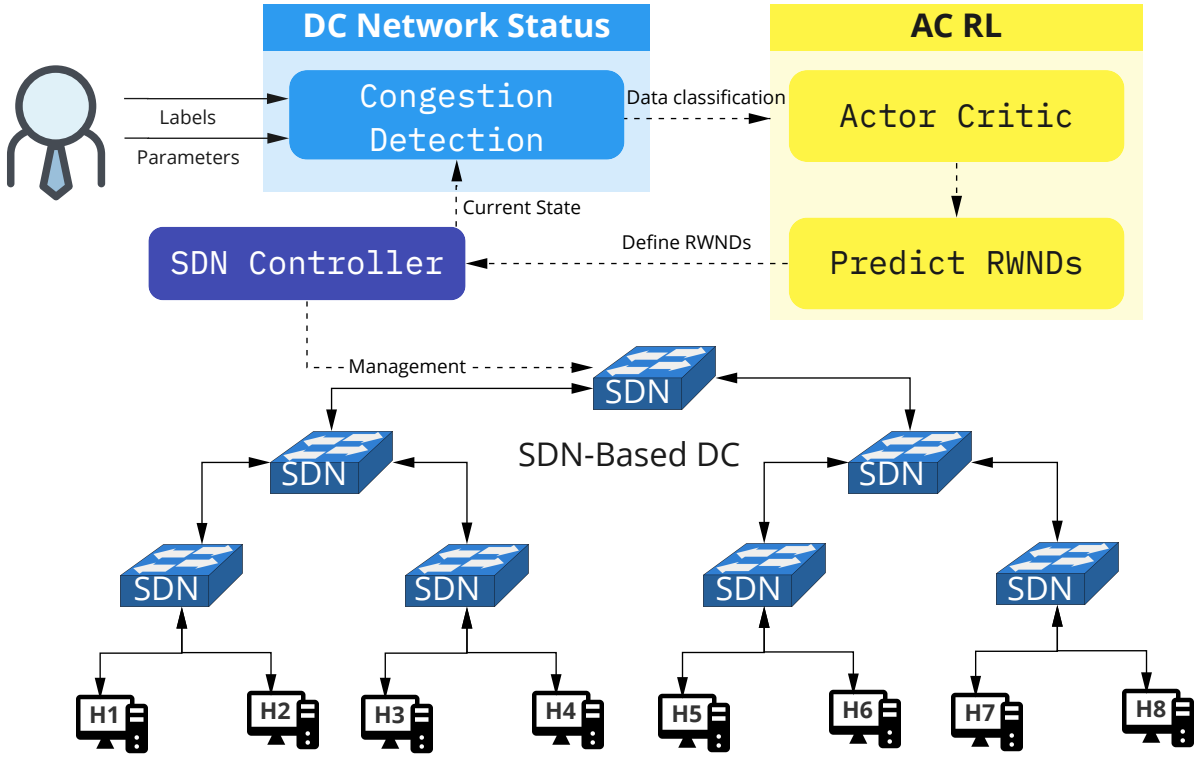
4.2.3 RSCAT modules

Figure 8 illustrates the management and network data flows of the proposed mechanism. Analyzing from the bottom-up, a set of servers (H1 to H8) are interconnected by SDN-aware switches (using OpenFlow 1.3 (FOUNDATION, 2012)). All of those switches are managed by a centralized controller (the SDN Controller). The controller is then running the RSCAT tool, and therefore, gathering monitored data from DC and applying TCP segments reconfiguration rules to the network. This is represented by the arrows coming in and out of the controller rectangle. Alongside the controller, RSCAT modules perform data classification and decision-making based on historical and ongoing network data on the AC RL frame.

4.2.3.1 DC Network Status

This module periodically retrieves data from the SDN controller for every switch port and flow, composing snapshots of the current network state. For each switch port, RSCAT gathers the number of packets, bytes, drops, errors and collisions (received and sent), and the relative difference of received bytes between the current read event and the last one ($\frac{receivedBytes}{lastReceivedBytes}$).

Figure 8 – General illustration of the management and network data flows. Source: author.



The rationale behind this equation is to obtain insights about the end-points attached to a given port, in other words, to identify whether the congestion control algorithm is increasing or decreasing its private CWND. In turn, for each flow, RSCAT accounts the number of packets and bytes transferred, as well as the maximum end-to-end available bandwidth. The Table 2 shows an example of data received by the module. All of the data comes directly from the network information except the columns “Byte Diff”, “Packet Diff” and “Relative diff”, which are calculated with the data from prior passes.

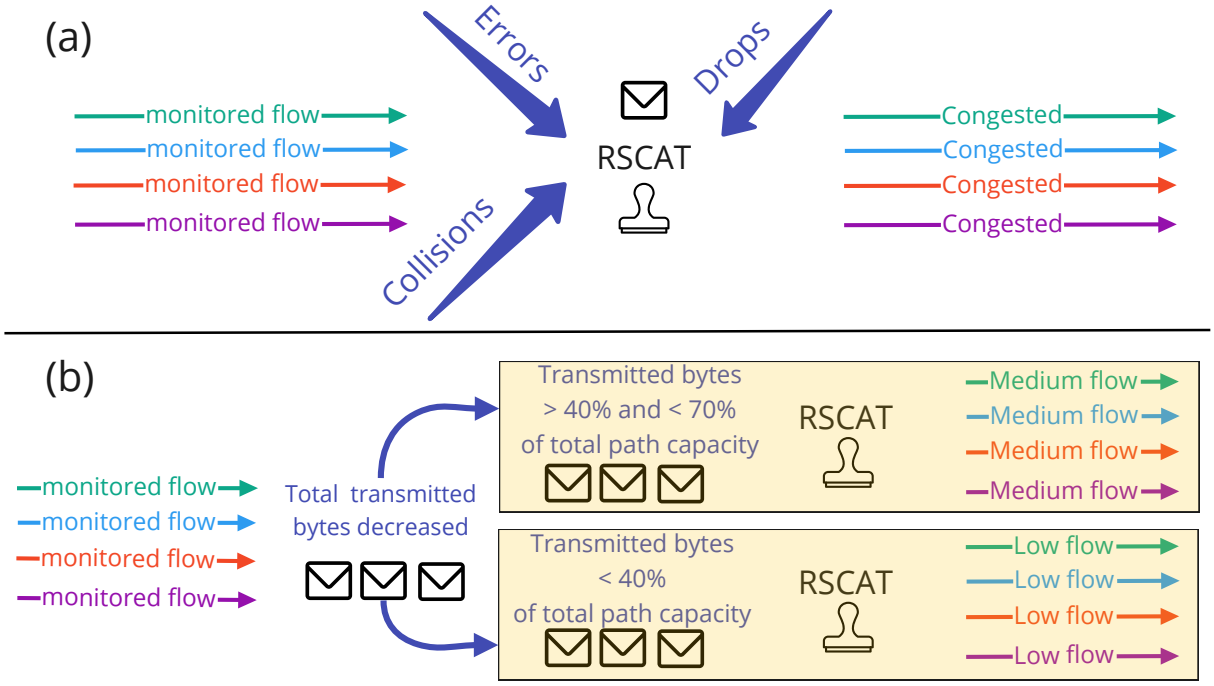
Table 2 – Example of data input that the first module receives. All of the data is originated from the network, some with some processing included.

Byte Diff	Packet Diff	RX Packets	TX Packets	RX Bytes	TX Bytes	RX Dropped	TX Dropped	RX Errors	TX Errors	Collisions	Relative diff
1088527845	19483	9635	11096	446610368	756324	0	0	0	0	0	0.03
1607960	23832	15577	16002	454105510	302974162	0	0	0	0	0	-0.03
0	0	20511	21534	580593326	329119969	0	0	0	0	0	-0.02
0	0	16048	15617	328203743	449025846	0	0	0	0	0	0.06
1780822	26455	15736	15134	303251548	433681424	0	0	0	0	0	0.1
4222022	61075	11100	9166	748104	597796356	0	0	0	0	0	-0.07
3177492	46942	15131	15738	433681226	303251688	1	0	0	0	2	0.16
573073451	3682	10168	11498	467205964	782928	0	0	0	0	1	-0.05

A supervised data classification algorithm is applied to detect ongoing DC congestion. Based on empirical analysis, the labels are classified into four main groups: **High**, **Medium**, and **Low** congestion probability, or **Congested** flows. The first three groups identify traffic intensity, with the last one indicating congestion. Although these labels are targeted at the ML algorithm, they do have a real-world meaning. The Congested label means that something on the network has flagged a network error (such as a drop), while the High, Medium and Low

labels mean that specific flow's throughput is high, medium and low, respectively and relatively speaking. In summary, errors, drops, collisions, and other switches and flow characteristics are combined with thresholds representing the network usage (e.g., 20%, 40%, and 70% of total end-to-end capacity) to apply labels for each flow. Figure 9 depicts the flow labels and their characteristics. Initially, all flows are classified in the High group. Latter, previously monitored flows are accounted as Congested, Medium, and Low, if necessary. Otherwise remains with the High label.

Figure 9 – Classification labels and action thresholds. Source: author.



As indicated by Figure 9(a), a flow is classified as **Congested** in the occurrence of errors, drops or collision events. These events come directly from the network through SDN. The other flows are classified according to the usage of the total available bandwidth for a given path (Figure 9(b)). Summarizing, flows with less than 40% of usage are classified as **Low** usage, between 40% and 70% as **Medium** while everything else is classified as **High** usage flow. The rationale behind this approach is to represent the impact of private TCP decisions regarding the increasing or decreasing of the segment's window. Upon finishing this classification, the AC RL module is invoked, receiving as input the same network data used to classify along with the classification results. With the algorithm that it uses, this problem can be classified as a classification problem, and uses classification techniques to solve it.

4.2.3.2 AC RL to configure TCP flows.

Following the data flow depicted in Figure 8, the decision process starts with the execution of the DC network data classifier, which invokes the AC RL module. When invoking the AC module, the classification module forwards the same input data it received along with it's own classification information. Table 3 shows an example of the data received by the module. It is important to point out that all of the data is originated or derived from the network, with the exceptions being the columns "Byte Diff", "Packet Diff" and "Relative diff", which are calculated by the controller and the "State" column, which are generated by the classification algorithm.

Table 3 – Example of data input for reinforcement algorithms. All of the data is originated from the network, some with some processing included.

Byte Diff	Packet Diff	RX Packets	TX Packets	RX Bytes	TX Bytes	RX Dropped	TX Dropped	RX Errors	TX Errors	Collisions	Relative diff	State
1780822	26455	15736	15134	303251548	433681424	0	0	0	0	0	0.1	Low
4222022	61075	11100	9166	748104	597796356	0	0	0	0	0	-0.07	High
3177492	46942	15131	15738	433681226	303251688	1	0	0	0	2	0.16	Congested
573073451	3682	10168	11498	467205964	782928	0	0	0	0	1	-0.05	Medium

Internally, the actor is responsible for using the base set of rules to learn how to respond to each scenario, receiving various inputs that it has never seen before and trying to get the appropriated answer (as discussed in Section 3.1.4.2). The actor selects the appropriate RWND for each connection, from a set of predefined options: 4 KB, 128 KB, 1 MB, 8 MB and 32 MB. Those values were chosen by selecting the most common default values in Linux-based operating systems for the default RWND and adding some steps in-between. After each action the actor takes, the critic judges them following some predetermined base rules, current state and previous one, giving the actor a score for its last action and learning how to better judge. The AC aims at maximizing the accumulated reward, which is account as depicted by Algorithm. 1. For each flow, the algorithm analyses the current throughput achieved from the latest RWND intervention, choosing a return value between $\{-10, -5, -2, 5, 10\}$ guided by the data classification thresholds and network status evolution. Theses reward values are strictly used by the ML algorithm, and were empirically defined. It is important to notice that both Actor and Critic are applied instances of the regression technique, as they both use past examples to predict future outcomes, or, in this case, predict the optimal solution. It is important to remember that this deterministic algorithm is just a starting point for the Actor-Critic. As it runs and learns, it'll generate it's own rules and reward values.

Specifically, *eql_threshold* is used to determine if two throughputs are similar (it has a value of 5%). The actor then uses this value to learn how well it has performed. It is important to note that in the following runs, the critic will also receive the difference of states as its input, so that it can also learn how to perform better. In the long-term, the training will then be optional, and both modules can then start to run in production mode, which will reduce training and disable validation to increase prediction and classification performance and reduce the time taken. Finally, the SDN controller is informed about all TCP flows that must be reconfigured

and their correspondent RWND configurations. At this stage, RSCAT can then intercept some TCP segments and reconfigure them with the new optimal RWND value so that the network end-points can use this value internally.

Algorithm 1: AC reward computation.

```
if current_throughput − last_throughput < eql_threshold then
    if last action was to lower RWND then
        reward is good(5)
    else if last action was to increase RWND then
        reward is somewhat bad(−2)
    end
else if current_throughput < last_throughput then
    if last action was to lower RWND then
        reward is very bad(−10)
    else if last action was to increase RWND then
        reward is very good(10)
    end
else
    if last action was to lower RWND then
        reward is bad(−5)
    else if last action was to increase RWND then
        reward is good(5)
    end
end
```

4.3 RELATED WORK

The related work presented here are examples of improving the TCP performance using various methods, even ML algorithms. Some of them use ML to classify a given connection, while others try to improve the TCP algorithm in some way. The Table 4 brings a compilation of the discussed works to facilitate the visualization and ideas of each work.

4.3.1 Traditional congestion control methods

In Alizadeh et al. (2010) the DCTCP is presented, an alternative algorithm for congestion control in TCP-based DCs. It uses network device guidance and information to achieve better congestion control performance. But, it requires the support of each network device, and while it does solve some problems and perform well under some scenarios, it fails to accomplish good results when dealing with multiple concurrent servers. In turn, in Cardwell et al. (2016) the BBR is presented, a congestion-based congestion control algorithm. It is a solution that patches the Linux kernel of the server and makes the BBR the default congestion control algorithm. This algorithm has shown to act better under certain situations where the traffic is in bad shape, as it's main focus is to act solemnly under congestion. The BBR algorithm, however, does require a change in the server's operating system receives a change, and it does not work using the network information, though it is designed with DCs in mind.

Zhang et al. (2019) propose the tool *Mystique*, which is an automated dynamic congestion control switching scheme tool that allows the network administrator to pick the desired congestion

control algorithm to be used in certain connections, allowing for the best one to be picked. It does not solve the congestion problem by itself, as it does aim to solve congestion, but instead, it chooses what the best algorithm would be. It does require a modification in the switch software, but not in the server or clients.

He et al. (2016) proposes the AC/DC TCP, a tool that enforces per-flow congestion control policies without making changes to the machine or network hardware by implementing an administration-defined algorithm over DC congestion control algorithms, like DCTCP.

Finally, Abdelmoniem e Bensaou (2021) presents a solution that works on either the Linux kernel or in the switches software. This solution is designed to allow the DC switches and routers communicate with each other and transmit the network state, and if needed, change the TCP's receive-window header fields to a pre-calculated value, in which the original untouched congestion control will perceive this as congestion and take action.

4.3.2 ML-based congestion control methods

In Abbasloo, Yen e Chao (2020b) and Abbasloo, Yen e Chao (2020a) it is described the project ORCA, also called DeepCC, which is a plugin for TCP connections that aims to increase the throughput and reduce latency of existing TCP congestion control algorithms. While it focuses on cellular connections, it does show improvements over other methods of congestion control, but, as it still uses the data available only in the end of each connection, it is still prone to some common problems of TCP congestion control, and this method does require some change in software for all computers on the network.

Both Babayigit e Ulu (2021) and Tosounidis, Pavlidis e Sakellariou (2020) present a deep learning-based load balancer for SDN networks. Both use deep learning methods to create a mathematical model that performs in a SDN network operating a load balancing algorithm. On Babayigit e Ulu (2021) it is compared with different ML methods, while Tosounidis, Pavlidis e Sakellariou (2020) focus on comparing against other known load balancing algorithms, such as Round-Robin.

In Chang et al. (2021) it is presented the MAGNet project, a ML-based network architecture that aims to deliver an endpoint in which the devices, that support it, may send a network request and, according to the data received, will apply some processing over the network packets. This is an application-aware tool, meaning that it analyses the type and needs of each connection, and answers the best strategies to be applied in the packets.

Eom et al. (2021) present a method for traffic classification, in which it uses the SDN architecture to obtain network traffic data and then runs it through a ML classification algorithm. This is used to classify the traffic and enable future actions to be taken upon certain traffic groups. In turn, in Balakiruthiga e Deepalakshmi (2021) it is proposed a traffic management tool coupled with an SDN network and DC-based. This tool uses reinforcement learning to choose which and how to apply the possible configurations: controller positioning, load balancing, routing or energy efficiency.

In Estrada-Solano, Caicedo e Fonseca (2019) it is presented the NELLY tool, which uses incremental learning along with SDN to identify elephant flows (or flows that carry a considerable amount of data), all while not creating more traffic in the network.

Fu et al. (2020) propose a tool that uses Q-Learning (a method of RL) to avoid congestion in SDN networks by applying forwarding rules to the network. But this project does only account for packet routing, and while routing might solve under some situations, the congestion control algorithms cannot be left aside.

In Bouzidi, Outtagarts e Langar (2019) it is proposed a network utility that, using reinforcement learning, predicts traffic state and tries to lower the overall network latency. It also uses SDN, but only uses the QoS utilities, which might not solve the congestion problem by itself. This tool also has no focus on DC networks, which is a requirement for our proposal.

Finally, Talpur (2017) presents a network tool that merges the capabilities of SDN with the power of ML to create an algorithm that detects congestion throughout the network. In the next suggested steps, it recommends to go beyond detection and apply rules to mitigate or prevent congestion.

4.3.3 Discussion

With all of these related works mentioned, it is clear that using ML methods for solving network problems, especially congestion control, is something that has been gaining popularity, and is actively being used in the academic and professional world. Some works deal with routing or forwarding of the packets, which could lead to some improvement relating to congestion control, but it is still not enough under some situations, so another approach is needed, like changing the behavior of already existing congestion control algorithms. Another thing to keep in mind is that some tools require the change of at least one of the nodes in the network, which could be a problem when the DC is comprised of heterogeneous connections, clients and software, so a solution that changes only network hardware or software is desirable. One last thing that could be taken from the presented related works, is that using SDN in conjunction with ML is a good idea, due to the data generated in a SDN controller and the fact that the ML algorithm could be hosted inside the controller, meaning that no changes in the hosts is necessary, and also the fact that SDN gives the algorithm more capabilities than a regular network switch, like QoS, changing the packet information or numerous other possibilities.

Article	Uses ML	Modify TCP	Solves congestion	Needs host or switch changes	DC focus	Uses SDN
Alizadeh et al. (2010)	No	No	Yes	Yes	Yes	-
Abbasloo, Yen e Chao (2020b)	Yes	Yes	Yes	Yes	-	-
Abbasloo, Yen e Chao (2020a)	Yes	Yes	Yes	Yes	-	-
Cardwell et al. (2016)	-	Yes	-	Yes	Yes	-
Babayigit e Ulu (2021)	Yes	-	No - Load Balancing	-	Yes	Yes
Tosounidis, Pavlidis e Sakellariou (2020)	Yes	-	No - Load Balancing	-	Yes	Yes
Zhang et al. (2019)	-	Yes	Maybe - Chooses best Algorithm	Yes	Yes	Yes
Abdelmoniem e Bensaou (2021)	-	Yes	Yes	Yes	Yes	Yes
Chang et al. (2021)	Yes	-	Maybe - Only applies application-based changes	Yes	Yes	-
Eom et al. (2021)	Yes	-	No - Just classification	-	-	Yes
Balakiruthiga e Deepalakshmi (2021)	-	-	No - Load balance and routing	-	Yes	Yes
Estrada-Solano, Caicedo e Fonseca (2019)	Yes	-	No - Just classification	-	Yes	Yes
Fu et al. (2020)	Yes	-	No - Just routing	-	Yes	Yes
Bouzidi, Outtagarts e Langar (2019)	Yes	-	No - Just QoS	-	-	Yes
Talpur (2017)	Yes	-	No - Just classification	-	Yes	Yes
He et al. (2016)	-	No - Creates new algorithm	No - Adds more control	-	Yes	No

Table 4 – A compilation of related works, if the use ML, if they modify the TCP header, if they aim to solve congestion, if they need to make changes to the hosts (servers and clients) or network switches if they focus on DC networks and if they use SDN.

4.4 CONSIDERATIONS

This chapter has brought the idea of the RSCAT, a tool for avoiding network congestion using data collected via SDN and the power of ML algorithms to allow better performance on already existing TCP congestion control algorithms. Enhancing known existing algorithms does not have the same drawback of implementing new ones where the support of new hardware is needed. Seeing that SDN is a common tool in DCs, taking advantage of its features makes a lot of sense, especially considering that most DCs with SDN isolate the links of the switch to the controller from the rest of the network, meaning that little to no disruption to the network is caused.

The idea of using a modular system allows network administrators to configure the tool for their own needs, being it only enhancing the congestion control algorithms, prioritizing a specific connection or what else it needs. It also means that training the Module 1 does not impact in any way the other Modules, so if it's detected that Module 1's classification is not great, it's possible to train it separately, and not waste any previous training data from other modules or time.

And finally, this thesis has presented 15 related works that have brought support to the elaboration of this proposal. Some have tried to enhance congestion control algorithms, like the DCTCP, a DC-specific congestion control algorithm. Other works have implemented network tools for predicting traffic or generating traffic rules with ML algorithms, such as the NELLY tool from Estrada-Solano, Caicedo e Fonseca (2019).

5 EXPERIMENTAL ANALYSIS

This chapter brings details about how RSCAT was tested. Mainly, in Section 5.1 it is discussed the proposed experimental environment and configurations, such as scenarios, configurations and methods used to perform the tests. Also, it brings some details about the metrics collected throughout the experiments. Section 5.2 organizes the collected results from our tests along with some deep analysis of them, validating that RSCAT has shown satisfactory results. Finally, Section 5.3 presents some considerations regarding the experiments, what they have shown and what the results imply.

5.1 EXPERIMENTAL INFRASTRUCTURE

Testing such complex tool as RSCAT would require a complex computing environment. Firstly, it requires some sort of computer network with a handful of computing nodes that interact with each other. Secondly, as tests progress and more ideas for experimental scenarios appear, it is interesting to have an architecture where the network topology is configurable. Those are the reasons that Mininet version 2.3.0 (LANTZ; HELLER; MCKEOWN, 2010) was chosen as the experimental network system. This version was the latest as of the writing of this thesis. Mininet allows for quick prototypes of complex network systems while being lightweight on the machine. Mininet's default switches also have support for OpenFlow 1.3, which is required by RSCAT. It also allows the network hosts to run on an isolated level, so that external programs running on the host computer does not interfere with results.

For the designed experimental scenario, two topologies were conceived: a simple tree topology and a FatTree topology. The simple tree topology (which is illustrated in the Figure 10) has a depth of 3 and a fanout of 2 (these are the values passed to Mininet's default tree topology generator algorithm). This simple tree topology was used for the test of each individual RSCAT module and one integration experiment. The FatTree (illustrated in the Figure 11) has a k value of 4, meaning that there are 4 core switches, connected to 4 pods (each containing 2 aggregation switches and 2 edge switches each) with all of the edge switches connected to a single computing node, called host. The FatTree topology does not come bundled with Mininet, so a custom Python script was written to generate it. This topology was used to run RSCAT on a DC-like environment as close to a real DC network as possible.

Although there are two topologies on the experiments, there are some similarities. For one, each link on the topology has a hard limit of 1 Gbps of bandwidth. It is worth mentioning that the whole infrastructure was hosted on a single computer, 8-cores, 16-threads i9 CPU with 48 GB of RAM. Another similarity is that all of the Mininet switches are SDN-compatible. They were all running version 1.3 of the OpenFlow (FOUNDATION, 2012) SDN protocol. On a side note, even though OpenFlow has newer versions (the latest version as of the release of this thesis is 1.6), version 1.3 has a higher market presence, and therefore, it is easier to have it supported on general DC hardware. Also, version 1.3 is the oldest version that has all of the necessary

Figure 10 – Simple Tree topology used for experiments. Source: author.

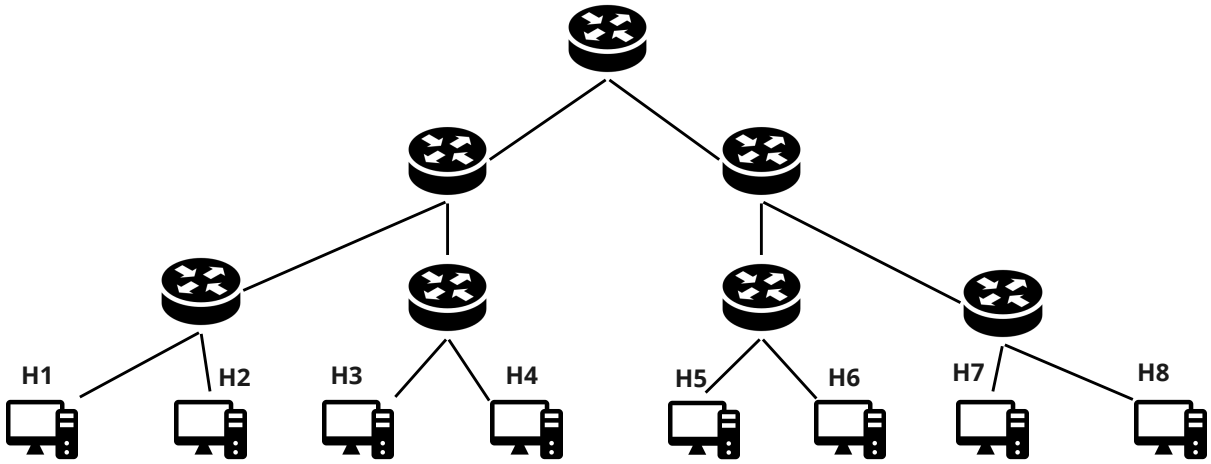
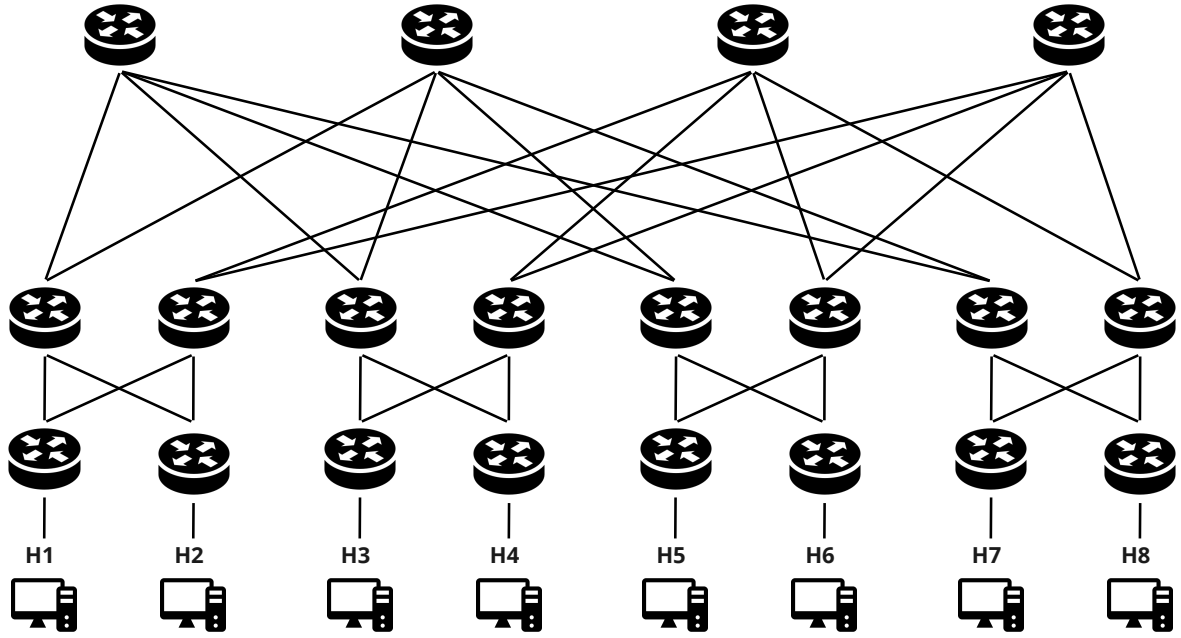


Figure 11 – Fat Tree (with $k = 4$) used for DC-like environment. Source: author.



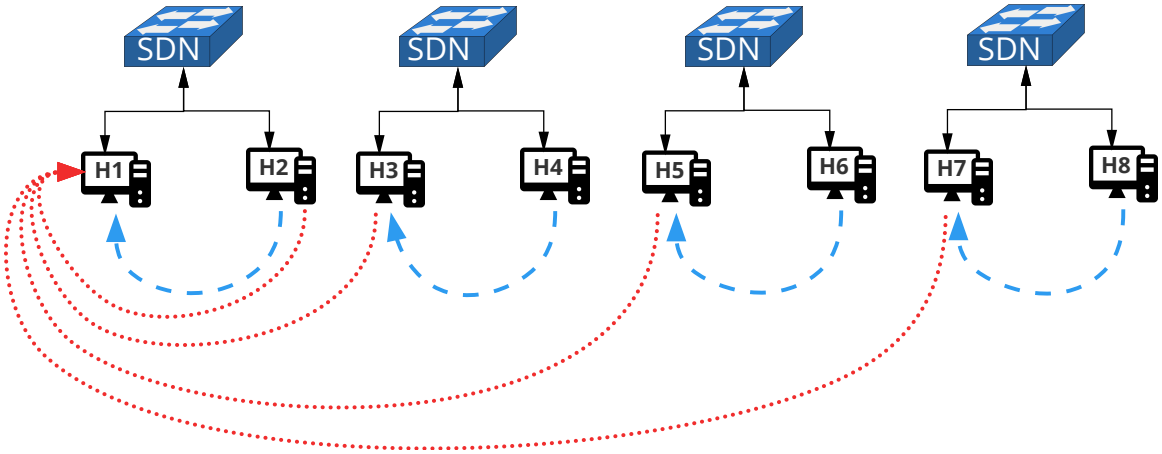
features required for RSCAT.

The algorithm for the AC is a custom-made on the Python programming language. It uses the hyperbolic tangent activation method and softplus method for standard deviation measurement provided by the Python library SKLearn (PEDREGOSA et al., 2011). The classification algorithm used is a custom one built over the Random Forest algorithm provided by the Python library SKLearn (PEDREGOSA et al., 2011). All of the activation and measurement methods used to come bundled with the PyTorch (PASZKE et al., 2019) library.

Another shared configuration between all of the experiments is how the network connections were configured in the application level. Figure 12 illustrates the connections for both topologies. The blue (or the dashed lines) represent the connections set up in order to not cause

congestion on the network. This specific setup was used when testing the individual modules for RSCAT. The other configuration is illustrated with the red (or dotted) lines, which represent a scenario where congestion is possible to happen on the network. And finally, every execution in every scenario was repeated 5 times to ensure no fluctuations or erratic results. For every single experiment and run, the controller was restarted, Mininet's cache was cleared and no application was running in the background, only the core OS processes.

Figure 12 – Connection directions throughout the experiment. Red (or dotted) is for congested scenario and blue (or dashed) is for non-congested scenario. Source: author.



5.1.1 Metrics

When dealing with computing networks, there are a couple of metrics that represent the connections' performance. One of the most basic ones is the throughput. Throughput is a metric that expresses the speed at which some amount of data can or has passed through a certain path. A higher throughput means that more data can be transferred in a certain frame of time. As an example, throughput of 1000 Mbps between A to B means that in one second, 1000 megabits can be sent from A to B. While usually the higher the better, this metric by itself does not mean much. This is because the throughput does not necessarily measure the factual limit of data transfer; it measures how fast the overall system can output on the time frame, so the file sizes also affect this measurement. With that said, in these experiments, the file sizes are higher than the maximum bandwidth, so this peculiarity can be neglected.

On a usual transfer, when no congestion is happening and all of the links are at their full capacities, the throughput approaches the smallest bandwidth on its path, which in these experiments is 1 Gbps. During congestion, fewer packets can transverse the network, and thus, less data is transferred (as seen in Section 2.1), which means that the throughput can be lowered. With that in mind, using the throughput as a measurement in the experiments can tell us how the congestion is affecting the network and how RSCAT is performing in assisting the congestion control algorithms. This means that, for the experiments in this thesis, the higher the overall

system throughput the better, as that means that the network usage is more effective. The objective of RSCAT for this measurement is to have the highest overall throughput on the system when compared to the other algorithms alone, meaning that it has successfully accomplished its task of assisting them in a positive manner. For example, if 4 connections go from having 1000 Mbps of throughput each to 1000, 1200, 900 and 950 Mbps respectively, the average network throughput has increased even so some of the connections have had their throughput lowered. This is because more of the network infrastructure is being in use.

Another metric that can help us diagnose RSCAT's performance is FCT. While FCT is somewhat self-explanatory, it is important to really understand what this metric means. On a computer network, most of the tasks are impacting people or businesses, which are both time sensitive. A task that takes more time to complete is usually less attractive, and this can be a deciding factor for the future of the service. So, having a good response time for the tasks is crucial in some systems, and that can be measured with FCT (DUKKIPATI; MCKEOWN, 2006). FCT is a measurement of how long a flow takes to be completed, and it is usually measured in seconds. Especially in this thesis, all of the FCT results have been transformed into a Cumulative Distribution Function (CDF), which is a graph where the X-axis represents a value (in the cases of this thesis, total time taken) and the Y-axis represents the probability of any random entry to be equal or less than the value on X-axis. This will help to visualize how the algorithm performs on the system as a whole.

A similar metric that to FCT is simply the time of an application to complete a task. This is one of the most basic performance metrics a program could have, but it shows how much time is needed for a certain application to run. This differs from FCT in that it considers the actual algorithm running, not only the data transfer. This metric will be used in the final scenario for the NAS MPB parallel benchmark tools, which output the time taken by themselves and no further action is required to process that result.

While this might seem redundant when compared to the throughput, this measurement is more user-focused, whereas throughput is more network-administrator-focused (DUKKIPATI; MCKEOWN, 2006). Having a lower FCT means that more tasks can be performed, or less time the network devices will waste on a single task. The objective of RSCAT for this measurement if to have a lower FCT when compared to other algorithms alone. If that is the case, it will have successfully accomplished its goal of assisting the other algorithms in managing congestion in order to turn the network more agile.

One last metric that is used in this thesis is not related to the network, but is aimed at the ML algorithm: classification rate. This measurement is a percentage of how many classifications were successful versus how many classifications were made in total. This is a common way of measuring classification algorithms' performance. Another important metric for ML that is more specific for this thesis and context is the time taken for each classification. This is a relatively simple metric which is just the measurement between the starting and finishing time of each classification, measured in seconds. While this applies to the classification algorithm, the AC

algorithm will be measured indirectly with the other metrics described in this section.

5.1.2 Scenarios

As there are a lot of metrics and components to be tested, several experiment scenarios have been selected to be executed. The Table 5 shows all of these scenarios, what they will execute, metrics collected, which topology the experiment was ran on, the algorithms being compared and the objective of that scenario.

Table 5 – Comparison of all of the experiment scenarios.

Scenario	Execution	Metrics	Topology	Algorithms	Objective
Detection of Network Congestion	Run Iperf with classification module alone	- Classification Rates - Time taken	Tree	N/A	Check module's performance when classifying network.
Reconfigurating TCP connections	Run Iperf with reconfiguration module alone	- Throughput - FCT	Tree	- CUBIC (RSCAT off) - DCTCP (RSCAT off) - CUBIC (RSCAT on) - DCTCP (RSCAT on)	Validate the impact when changing TCP's buffer during connection.
Integration Experiment	Run Empirical Traffic Generator (ETG) tool with full RSCAT	- FCT	Tree	- CUBIC (RSCAT off) - DCTCP (RSCAT off)] - DCTCP (RSCAT on)	Validate the entire tool's performance when using all modules.
Data Center-like Network Scenario	Run ETG tool with full RSCAT	- FCT	FatTree	- CUBIC (RSCAT off) - DCTCP (RSCAT off)] - DCTCP (RSCAT on)	Validate the tool's performance when running on a DC-like environment.
NAS NPB Benchmark Tools	Run NAS NPB tool with full RSCAT	- FCT	FatTree	- CUBIC (RSCAT off) - DCTCP (RSCAT off)] - DCTCP (RSCAT on)	Validate the tool's performance when running with programs similar to ones found on a DC environment.

Starting with "Detection of Network Congestion", this is a simple experiment which aims to check how the classification module performs when detecting the network congestion. In this scenario, only the classification module will be executed, so that we can measure its performance on a best case scenario, where there are no other activities happening on the network or other host nodes. This experiment will measure both the classification rates and time taken for each of the tested algorithms: RF, Support Vector Machine (SVM) and Multilayer Perceptron (MLP). As it is not dealing with congestion control, only congestion detection, no congestion control algorithms are being compared. The experiment is divided in 2 settings, the first follows the blue (or dashed) lines as seen on Figure 12 which are the pairs of hosts within the same local network (H1 to H2, H3 to H4 and so on) which was designed to not have network congestion. The other setting is following the red (or dotted) line, which are the hosts H2, H3, H5 and H7 connecting simultaneously to H1, to which the goal is to induce network congestion. It is important to remember that although the settings were different, they are considered the same execution as we are validating the classification of all states (Congested, High, Medium and Low, as seen on Section 4.2.3.1).

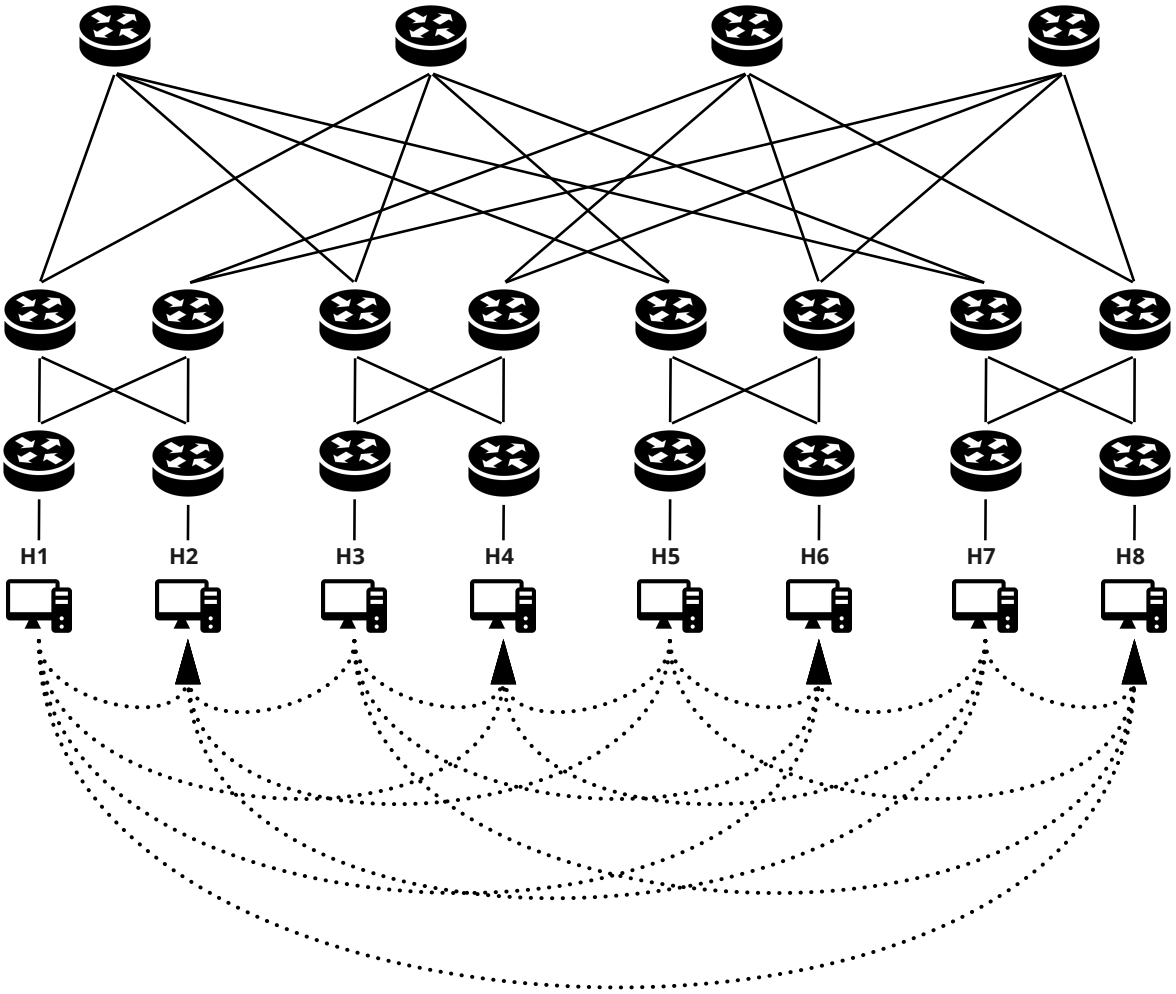
The "Reconfigurating TCP connections" experiment follows the same principle, which is to have a best case scenario where only a specific module is being tested in order to validate its performance. In this experiment, the tested module is the one designed to output a optimal TCP buffer for reconfiguration. This time, we compared the CUBIC and DCTCP algorithms with and without RSCAT running, and it was all executed in the Tree topology. The metrics collected from this test were the throughput and FCT, two popular metrics for network traffic.

"Integration Experiment" is the first experiment which executed both modules simultaneously in order to validate whether the solution as a whole works. Using a Tree topology, this experiment runs the ETG tool on hosts H2, H3, H5, H7 which keeps transferring files to the server H1 with random sizes, defined by a CDF configuration file, which range from 100 KB to 50 MB. ETG is a tool which tries to generate network traffic similarly to a DC. This experiment will compare the algorithms CUBIC vs DCTCP vs DCTCP with RSCAT on. ETG can represent real DC traffic with distinct flows size (mice and elephants), guided by traffic patterns described as CDFs. Multiple pairs of ETG client-servers were instantiated atop the tree-based topology depicted by Figure 12 executing the ETG input configuration proposed by (MENTZ; LOCH; KOSLOVSKI, 2020), which summarizes the DC traffic described in (BENSON; AKELLA; MALTZ, 2010). It is important to establish that DCTCP was chosen to run alongside RSCAT because of two factors: DCTCP is DC focused and it has shown greater performance when compared to CUBIC with RSCAT on. On this experiment, the FCT is used as metric.

Then, "Data Center-like Network Scenario" is a all-in-one experiment that proposes a similar experiment scenario that "Integration Experiment" proposes, with the exception that the topology and the ETG behavior have changed. This scenario has the goal of validating the performance of the whole tool in a scenario that has more similarities with a real life DC environment, and so, the FatTree topology was deployed in this scenario (illustrated on Figure 11). Also, as DC has tremendous amount of parallel traffic, the ETG tool was setup to send randomly from any target (H2, H4, H6 and H8) to any server (H1, H3, H5 and H7) at random orders and sizes (this behavior is illustrated on Figure 13, where the lines show the connection links). Other than the topology and ETG targets, everything else is the same: CUBIC vs DCTCP vs DCTCP with RSCAT on and using the ETG tool to run the benchmark. And once again, FCT was the selected metric to be measured.

Finally, following the same idea as "Data Center-like Network Scenario", "NAS NPB Benchmark Tools" is also another all-in-one experiment which focuses on having an even more real-to-life experiment. It does so by running the NAS Parallel Benchmark (BAILEY et al., 1991), which is a set of high-performance computing algorithms that run across a computer network. It sets a cluster of computing nodes across the network to run the algorithms, one by one. Each program as its own purpose, but all of the run across the network, transferring data across each other and running heavy calculations. This all is in order to have a set of tools that simulate more closely a cloud computing server running its applications. This scenario runs on the same FatTree topology as in the other scenario (illustrated on Figure 11, with H1 being the coordination node and H2, H4, H6 and H8 being the computing nodes. This tools generate the time taken to run the algorithms, so the result will be collected in seconds, so a lower value is better. The NAS algorithms that were executed were: BT, CG, EP, FT, LU, MG and SP, compiled locally using GCC on Ubuntu, using the class B configuration flag. The tools themselves were executed using the OpenMPI version 4.0.3 for AMD 64.

Figure 13 – Configuration of ETG targets for DC-like scenario. The ETG connections are displayed in dotted lines. Source: author.



5.2 EXPERIMENTAL RESULTS

This Section presents the results obtained by running each scenario. It also contains some discussion ideas about what scenario RSCAT seems to work best and what could be improved. Subsection 5.2.1 contains the results and discussion about the “Detection of Network Congestion” scenario. Subsection 5.2.2 presents the results and discussion about the scenario “Reconfigurating TCP connections”. Subsection 5.2.3 brings the results and some discussion about the “Integration Experiment” scenario. The scenario “Data Center-like Network Scenario” is presented in Subsection 5.2.4. Finally, Subsection 5.2.5 contains the results and discussion about the scenario “NAS NPB Benchmark Tools”.

5.2.1 Detection of Network Congestion

In order to test the classification algorithm, two scenarios were designed. The first was running Iperf for each close pair, as in, host H1 connects to host H2, H3 to H4 and so on. This

scenario is setup in a way so that no network congestion is generated, as every link will be operating at full capacity without interruptions. In the second scenario, H2 was the server, and H1, H3, H5 ha H7 would all connect to H2, in which a lot of traffic is bottle-necked when reaching H2, causing congestion.

In both scenarios, the execution was the same: start-up Mininet topology, start the Iperf servers and get the Iperf clients to send data to the designated server. As the topology had finished booting up, the controller immediately started capturing data and sending it to a custom script that would then save the data for later use. In order to extract as much data as possible, it kept collecting data for 1 hour. Finally, a script was run in order to analyze line by line in a slow old-fashioned way with custom parameters to label each state as congested or not.

After both runs, Module 1 was fed both collected and labeled data and started training and testing. The classification algorithm used is the RF provided by the Python library SKLearn (PEDREGOSA et al., 2011). Were a little over $500,000$ rows of data collected, and about $100,000$ were used for testing, selected randomly. The final result was a classification rate of over 98% of the collected states reserved for testing.

In order to compare with other known algorithms' performance in such an environment, two other techniques were also tested. SVM, just like RF, is a common name in the field of data classification with ML algorithms. This work used the one provided by SKLearn (PEDREGOSA et al., 2011). The other algorithm is the MLP, which is a bit different from the two previous ones. It is an Artificial Neural Network (ANN)-based algorithm, and it works by mimicking the brain, with its convoluted neural network. The comparison is in Table 6. It is important to know that each test was executed 5 times, each with the exact same configuration and dataset. It is possible to notice that even though RF and SVM had similar classification rates, SVM took longer to reach an answer, and time is of the essence in an environment like computer networks. This means that RF had the best overall performance by the time of classification.

Table 6 – Table with the results on the classification tests. RF and SVM have the higher rates, but SVM takes more time.

Algorithm	**Classification Rate**	**Time to classify**
Random Forest	$98\% \pm 2\%$	7 milliseconds $\pm$ 4ms
SVM	$97\% \pm 3\%$	1760 milliseconds $\pm$ 382ms
MLP	$64\% \pm 6\%$	4 milliseconds $\pm$ 2ms

5.2.2 Reconfigurating TCP connections

This scenario highlights the advantages and drawbacks of applying RSCAT with two traditional TCP variants: CUBIC (HA; RHEE; XU, 2008) and DCTCP (ALIZADEH et al., 2010), both running with default parameters. It is worthwhile to reiterate that RSCAT is applied as a complementary tool used together with traditional TCP algorithms. In this experiment, while the network is going, RSCAT periodically reconfigures the ongoing flows as it seems fit. The

results of each set of flows are individually presented in Figure 14 as CDF and boxplot graphs, for FCT and throughput metrics, respectively. The results are organized per host.

The first thing to notice is that for both FCT and throughput results (Figures 14a through 14h), RSCAT, as a complementary tool, has no noticeable negative impacts on the overall network performance. Specifically, Figure 14c and Figure 14d indicate that RSCAT decreases the FCT for flows crossing multiple switches, while only the combination with CUBIC can be beneficial for closest pairs (Figure 14a and Figure 14b). In Figure 14a, while RSCAT has had some inferior results during some parts when using the CUBIC algorithm, the overall network FCT has increased. In Figure 14b, RSCAT had no noticeable impact when used with CUBIC, but had a better performance on every instance of the tests; possible to see as the green line (DCTCP with RSCAT) is always over the red line (DCTCP without RSCAT). Both Figure 14c and Figure 14d show that RSCAT always had a lower FCT when compared to not using it.

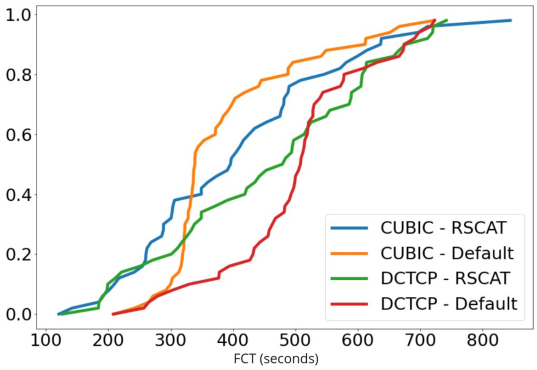
For the throughput, Figures 14e through 14h indicate that RSCAT offers higher throughput when compared to CUBIC alone and a similar result when compared to DCTCP. Specifically, Figure 14e shows that the CUBIC average had increased from an average of 217 Mbps to 402 Mbps for CUBIC, while DCTCP went from 284 Mbps (without RSCAT) to 301 Mbps (with RSCAT). Figure 14f shows that the average changed from 191 Mbps to 210 Mbps for CUBIC, while DCTCP both with and without RSCAT stayed at an average of 187 Mbps. Figure 14g shows that the CUBIC average changed from 205 Mbps to 307 Mbps, and 214 Mbps to 298 Mbps for DCTCP. Finally, Figure 14h shows that the CUBIC average hovered near the 200 Mbps for both settings while for DCTCP it increased from 201 Mbps to 276 Mbps. It is worthwhile to mention that RSCAT has improved performance in both FCT and throughput perspectives in several cases.

In summary, this scenario analyzed the performance of RSCAT based on elephant flows. When running against CUBIC alone, RSCAT has shown that about 50% of the cases have lower FCT, and 35% were similar. The worst performance was for host H2 (the nearest to the server), in which about 45% of the cases had higher FCT. For host H7, on the other hand, 80% had lower FCT. When comparing against DCTCP, RSCAT has shown about 82% of the cases have lower FCT, and 15% were similar. The worst performance was, again, for host H2 (the nearest to the server), where about 6% of the cases had higher FCT. For all the other hosts, more than 90% of the cases had lower FCT.

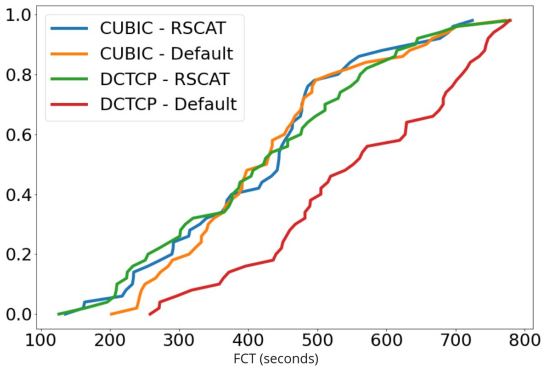
5.2.3 Integration Experiment

Table 7 organizes the FCT results according to flow size intervals: sizes smaller than 1 MB, between 1 MB and 10 MB, and between 10 MB and 50 MB. In addition, four sub-columns for each main column are presented: Q1, Q2, and Q3 being the first, second, and third quartiles; and the average (Avg). When transferring smaller flows (up to 1 MB), RSCAT has shown performance similar to DCTCP, with both having higher FCT values than CUBIC (this is highlighted in the table on the first main column (less than 1 MB) in the Avg sub-column).

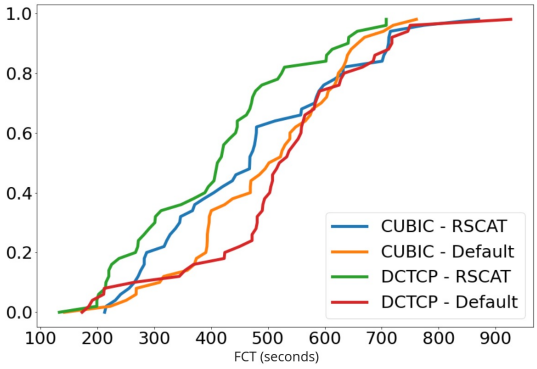
Figure 14 – Results: FCT (in seconds) and throughput (Mbps). Source: author.



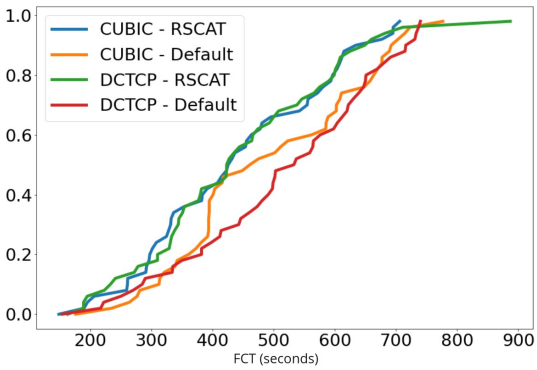
(a) FCT: H2 to H1



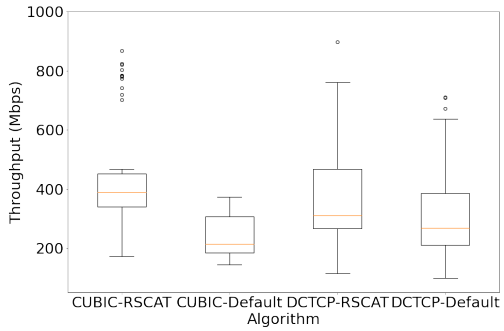
(b) FCT: H3 to H1



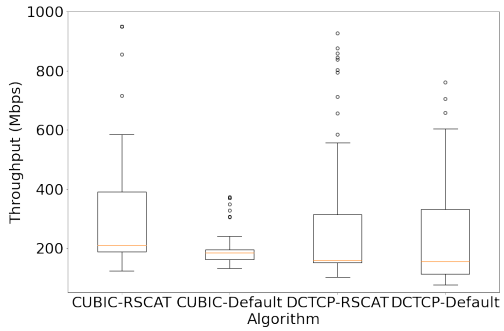
(c) FCT: H5 to H1



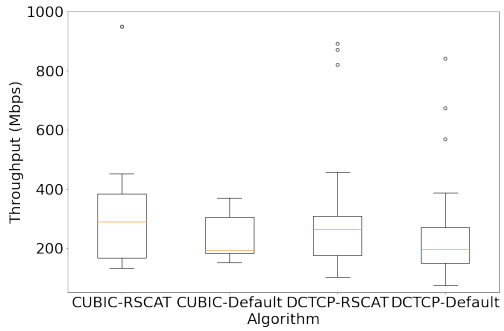
(d) FCT: H7 to H1



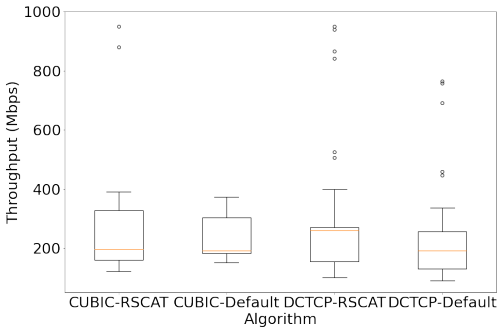
(e) Throughput: H2 to H1



(f) Throughput: H3 to H1



(g) Throughput: H5 to H1



(h) Throughput: H7 to H1

The difference between default CUBIC and DCTCP with RSCAT is substantial: approximately 35% (from 420 seconds on CUBIC to 640 seconds on RSCAT). As the flow sizes increase (as highlighted using a gray background in the main column in the middle), the behavior changes and default CUBIC comes to have the highest FCTs, and, once again, DCTCP and RSCAT perform similarly, and we see a consistent 15% decrease in FCT. For the most extensive flows (highlighted in the last main column), it is noticeable that RSCAT improves the performance of both traditional TCP variants, with FCT decreasing by approximately 10% throughout the experiments.

Table 7 – Results of FCT (in seconds) for ETG DC workload simulation.

		$x \leq 1$ MB				1 MB $< x \leq 10$ MB				10 MB $< x \leq 50$ MB			
		Q1	Q2	Q3	Avg	Q1	Q2	Q3	Avg	Q1	Q2	Q3	Avg
H2	DCTCP	150.70	395.10	752.00	603.84	4412.90	6283.35	9546.33	7838.49	70226.95	89825.20	121402.10	99655.41
	CUBIC	141.35	278.74	529.10	426.37	5066.66	6901.13	10180.28	8393.48	85121.33	104584.48	132124.03	113276.23
	RSCAT	155.40	413.20	799.40	640.08	4072.36	5883.36	8843.86	7319.61	62705.57	81085.15	108802.60	89907.21
H3	DCTCP	154.90	400.40	762.90	610.53	4444.60	6375.50	9451.23	7876.38	72584.15	94302.85	127743.00	105291.33
	CUBIC	143.00	278.85	526.19	420.99	5087.09	6949.47	10212.76	8370.94	82389.48	101322.87	131355.24	110851.32
	RSCAT	155.50	413.80	810.55	647.65	4061.84	5840.78	8797.08	7247.75	64942.42	84891.18	117156.35	95738.18
H5	DCTCP	153.50	397.60	771.20	610.48	4405.02	6297.65	9484.83	7821.14	71939.68	94700.10	131875.68	106427.02
	CUBIC	141.24	277.53	526.68	418.65	5039.87	6876.43	10127.04	8315.65	83385.72	101379.80	132694.54	111831.58
	RSTCAT	158.90	416.40	803.20	641.38	4059.93	5823.60	8658.44	7209.41	68845.54	88888.21	121075.16	98687.04
H7	DCTCP	156.20	402.40	769.65	615.12	4446.70	6356.80	9481.30	7870.41	73620.78	93732.40	129631.65	105999.08
	CUBIC	141.57	277.09	525.80	423.28	5088.60	6890.51	9970.18	8269.16	81961.77	103564.01	130931.57	113174.28
	RSCAT	156.58	413.20	811.93	644.54	4081.08	5890.47	8847.02	7294.58	68868.48	88131.45	114725.71	97332.72

5.2.4 Data Center-like Network Scenario

To determine the performance of RSCAT in a DC-like environment, we resorted to running a similar test seen in Section 5.2.2 but on a more common DC topology. Table 8 organizes the FCT results in the FatTree topology. As with the previous experiment, it organizes the results according to flow size intervals: sizes smaller than 1 MB, between 1 MB and 10 MB, and between 10 MB and 50 MB. The four sub-columns for quartiles and the average for each main column are also present in this experiment. The results follow the same high-level behavior as in the previous results: RSCAT shows similar performance to DCTCP on smaller flows (less than 1 MB) and starts to show an increase in performance (as seen as the number of seconds decreases) as the flow size grows. For ease of reading the results, all of the Avg columns of host H3 have been highlighted to show the difference RSCAT has when the flow size changes, compared to the other algorithms.

Specifically, when comparing with mice traffic, RSCAT has shown the worst performance (an average of 633 s vs. 571 s and 379 s of DCTCP and CUBIC, respectively). As the file sizes increased, the scenario changed considerably. On the transference of H3, RSCAT managed to achieve an average of 11.468 s while DCTCP and CUBIC got averages of 20.175 s and 19.405 s, respectively, decreasing the FCT by almost half. The largest transfer files had similar results. For H3 again, RSCAT, DCTCP and CUBIC got respectively 86.847 s, 129.838 s and 131.461 s.

Table 8 – Results of FCT (in seconds) for ETG DC workload simulation on a DC-like topology (FatTree)

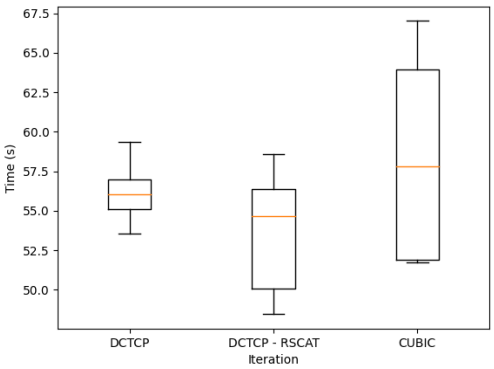
		$x \leq 1$ MB				1 MB $< x \leq 10$ MB				10 MB $< x \leq 50$ MB			
		Q1	Q2	Q3	Avg	Q1	Q2	Q3	Avg	Q1	Q2	Q3	Avg
H2	RSCAT	159.61	392.76	791.85	637.82	4373.89	10191.86	20347.34	17813.07	61224.74	74776.43	92839.06	98225.43
	CUBIC	149.4	372.76	646.61	412.67	5097.51	13040.09	22912.67	10622.96	84668.21	106583.75	144025.94	143080.25
	DCTCP	145.88	382.69	746.28	570.59	4966.36	12178.12	24639.3	20275.95	75748.94	88519.73	135856.49	125310.45
H3	RSCAT	154.32	387.21	742.67	633.76	4656.85	11721.47	20440.22	11468.97	66797.09	83888.04	100381.19	86847.58
	CUBIC	158.57	393.71	641.29	379.8	5164.75	12804.9	25811.83	20175.69	82727.48	92072.38	144367.45	129838.99
	DCTCP	144.2	378.88	743.96	571.46	4761.55	12543.64	25190.23	19405.28	78696.68	89405.62	132503.65	131461.47
H5	RSCAT	144.7	375.28	718.38	577.41	4674.77	10844.04	21599.36	18515.71	74505.7	88315.79	114085.56	116642.54
	CUBIC	157.56	403.53	727.88	431.44	5076.58	11869.82	26161.69	21198.38	80963.42	104984.84	117806.24	110741.65
	DCTCP	160.47	430.04	840.91	647.56	4363.39	11697.24	19538.35	10347.62	85034.18	95272.39	142044.66	121018.23
H7	RSCAT	150.55	400.28	723.0	597.04	4379.35	11344.79	22853.51	18314.3	70661.61	89915.12	102969.85	96032.37
	CUBIC	142.99	350.25	692.05	374.05	5126.92	12199.76	22889.82	13287.93	87672.7	107332.63	140301.78	143168.82
	DCTCP	141.61	379.99	676.66	554.98	4560.91	11862.2	23049.34	19149.33	83775.65	101150.71	136416.53	135959.16

5.2.5 NAS NPB Benchmark Tools

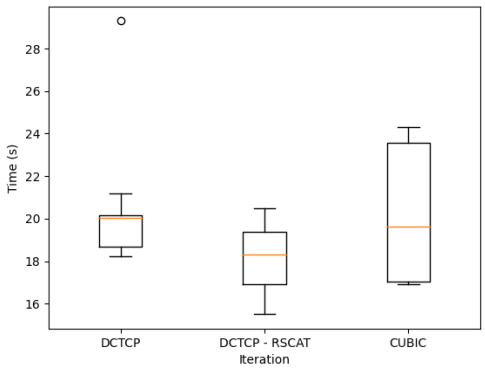
Each NAS algorithm ran individually, and to avoid any possible caching interference, we first ran all of the algorithms before starting to repeat the run. Running the NAS NPB Benchmark tools has yielded 7 results illustrated in Figure 15a, Figure 15b, Figure 15c, Figure 15d, Figure 15e, Figure 15f, and finally, Figure 15g. All the graphs follow the same rules: a boxplot showing the time taken in seconds to complete the problems for CUBIC, DCTCP and RSCAT with DCTCP. As seen in the results, the RSCAT has increased its performance in several scenarios. For the BT problem, RSCAT has an average of 53.59 ± 3.46 seconds, followed by DCTCP with 56.27 ± 1.76 seconds and then CUBIC with 58.42 ± 6.34 seconds, showing a shy improvement over both algorithms. For CG, the results were 18.24 ± 1.58, 20.53 ± 3.06 and 20.22 ± 3.25 for RSCAT, DCTCP and CUBIC respectively, again showing some minor improvements in the executions time, all in seconds. On the EP program, the average times were RSCAT with 3.73 ± 0.21, DCTCP with 3.78 ± 0.22 and CUBIC with 3.96 ± 0.18, showing minimal differences in all the algorithms. On the FT program, the average times for RSCAT were 13.13 ± 1.37, for DCTCP were 12.80 ± 0.52 and for CUBIC they were 13.47 ± 1.72, and once again, showing minimal differences. On the LU program, the average times were RSCAT with 36.57 ± 2.49, DCTCP with 39.29 ± 3.05 and CUBIC with 39.60 ± 4.08, showing an improvement of about 3 seconds over the other algorithms. The MG algorithm also has not shown significant differences, with an average for RSCAT of about 2.61 ± 0.23, for DCTCP of 2.96 ± 0.34 and the average for CUBIC of 2.89 ± 0.52. Finally, for the SP problem, RSCAT got an average of 97.00 ± 7.48, DCTCP got 103.24 ± 5.30 and CUBIC achieved 107.16 ± 17.00, showing a more considerable differences between RSCAT and the other algorithms, though not by large.

To understand the results better and why some algorithms had more difference than others, it's important to understand what each algorithm is doing behind the scenes. EP is the algorithm that has minimal network usage, so RSCAT could not improve its performance as there was no network connections to enhance (MARCONDES et al., 2016). A similar effect happens to FT, which has an insignificant network impact. This behavior can be seen on the boxplot as the averages are very close to one another. MG, LU, SP and BT are algorithms with higher network usage. Each has its own signature, like SP has a high volume, while the others have a lot of tiny packets being shared (MARCONDES et al., 2016). There algorithms with higher network usage have shown to benefit from the usage of RSCAT. The change in performance is visible through the boxplot, in which RSCAT has a lower average time (the red line in the middle of the rectangles), and usually a lower minimal point. This scenario has shown that RSCAT takes advantage of long-running traffic, where shorter problem instances have little to no advantages, but longer-running algorithms could increase their performance and execute in lower time compared to when not running RSCAT.

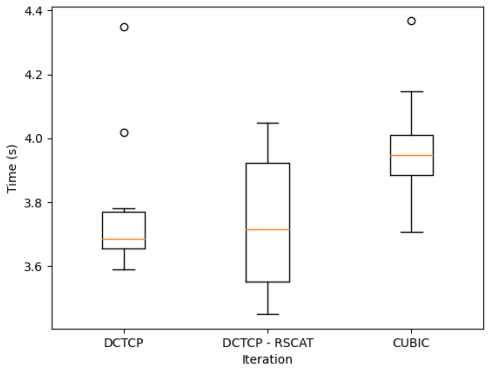
Figure 15 – Results: Time (in seconds) for the execution of several NPB problem. Source: author.



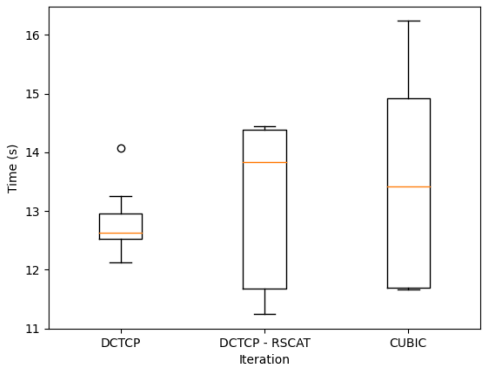
(a) BT Program



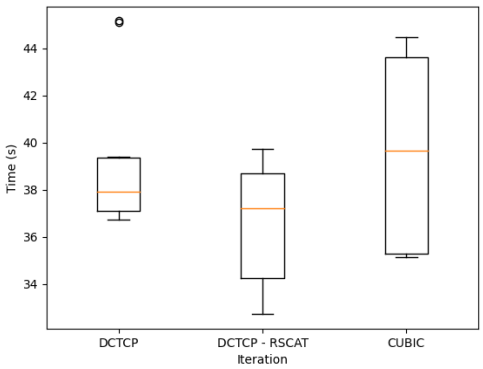
(b) CG Program



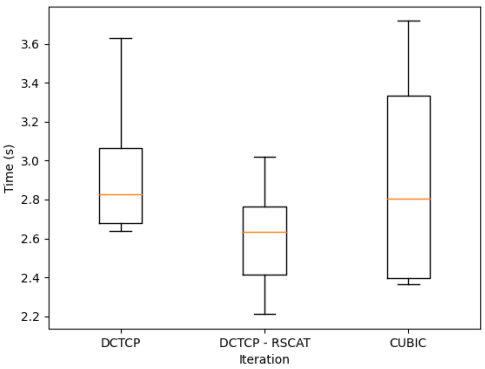
(c) EP Program



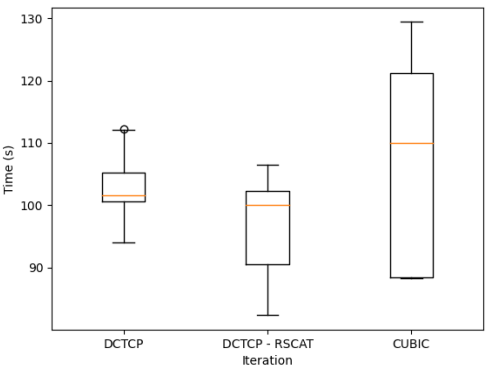
(d) FT Program



(e) LU Program



(f) MG Program



(g) SP Program

5.3 CONSIDERATIONS

The first two experiments, which have executed both modules on their own, had the focus on testing each module and validating if they worked as expected. The results speak for themselves, as both of them were positive. In the classification experiment, it was shown that RF had the best relation between classification rate versus time taken. It is extremely important for a network-related (especially a congestion control-related) algorithm to be fast and agile, so as to respond as quickly as possible to network changes. And that experiment has proven that it is indeed possible for the module to be applied on a network, taking as little as 7 ms to classify while maintaining a classification rate of over 96%.

The second experiment which tests the reconfiguration module alone has also had positive results. The module has not decreased once the performance on the network, but when it did impact it, it impacted positively. When running, the FCT was reduced in 35% to 90% of the cases when running RSCAT. And even though the throughput was lower in some cases, the overall FCT was decreased by having fewer re-transmission packets sent when the network is assumed to be congested.

And by running RSCAT on a network topology common in DC (the last two experiments), with data traffic generated by a tool that is specifically designed to imitate DC traffic, it's safe to assume that RSCAT has shown great potential. It is also important to remember that although it has been some decrease in FCT on some results, the overall network performance has not decreased in any case where the transferred files were larger. Thus, with all of the collected results, it's safe to say that RSCAT has checked all of its functional requirements: it detects the congestion when it happens using SDN, it does generate network rules and it does not impact the network's performance negatively. The experiments have also shown that RSCAT is not suited for all kinds of traffic. As seen in Subsection 5.2.3 and 5.2.4, when dealing with small transfer files, RSCAT had the worst performance, and as the file sizes increased, so did the RSCAT's performance, achieving the best among the tested algorithms. This behavior is expected with the current implementation, which takes some time to act and re-apply the network rules. Also, close pairs have shown to have a decrease in performance by RSCAT, and as the distance between server and client increased, so did their performance when running with RSCAT.

Another crucial fact is that throughout every scenario and experiment, the controller's CPU and system memory usage was measured. This is an important statistic as it shows how efficient the RSCAT algorithm is and how resource-intensive it behaves. But the experiments have shown minimal impact. During runs, the average CPU usage was hovering over 5%, with spikes up to 20% when new flows were initiated. RAM usage was also a non-issue, as it was fixed at 336 MB during every experiment. Also, another important note is that this benchmark, although it has had some tools to simulate a DC-like environment is still not, in fact, a DC. The behavior was tested based on simulations, and thus the performance on real-world DCs might be different. And RSCAT is only a prototype, a POC to show that the ideas of the thesis are valid,

but there are space of improvement on its code and architecture.

6 CONCLUSIONS

The existence of congestion in computer networks is a threat to every application that is connected to another computer. Dealing with this threat is a necessary, but a hard challenge. Many algorithms come up with new and more advanced solutions for congestion control, but most of them rely on the information of a single client-server pair, the TCP congestion control algorithms. To gather more information to produce more efficient algorithms, the network-aided congestion control algorithms were created. But the common problem for them all is the lack of support, as each hardware manufacturer must adapt and support each new algorithm, which is a hard task for both the manufacturer and the data center administration.

Using the information SDN provides is an easier task, as its employment is far greater, as many data centers have upgraded their hardware to support SDN. What makes this process easier is that SDN has one main protocol, OpenFlow, and this allows the manufacturers to only worry about a single protocol, instead of many.

This thesis proposes RSCAT, a tool that uses information available in SDN controllers to enhance current existing TCP congestion control algorithms. RSCAT uses SDN features, which makes it possible to process many network traffic data gathered by the controller, and apply them back to the network as network rules that can change the traffic behavior to adjust to the best efficiency on the current hardware. RSCAT makes use of ML algorithms to predict the best possible window size for each connection, so that the SDN controller may alter the TCP packet header to update the RWND value. This change will make whatever congestion control algorithm updates, temporarily, the CWND value, thus, controlling the throughput of that connection in order to reach the network's desire, which can be configured by the administrator.

The collection of results has shown that when applying RSCAT along with existing congestion control algorithms like CUBIC or DCTCP, the overall network performance has either remained the same or increased. Testing the modules alone has proven their effectiveness, when the integration test has shown that the modules cooperate to create a tool that, when in use, alters traffic rules according to the network state. As it trained more and more, it started to generate better rules which, at some point, have resulted in better performance when compared to not using RSCAT. The DC-like scenarios have also shown that RSCAT may be suited for running in a DC environment, as both ETG and NAS Parallel Benchmark tools generate traffic that matches that of various real-life DCs, everything running inside a simulated FatTree topology, a common one in DCs. Not everything is perfect though, as RSCAT shows some weaknesses when dealing with mice traffic or when the server and the client are near each other. But when dealing with elephant traffic, and the host is not side-by-side, RSCAT has shown great results, in some tests reaching half of the FCT of other algorithms.

During the development of this thesis, some questions have arised, and can be explored in future projects. One of them is to improve RSCAT's dataset, by training with real-world scenarios or by replacing the machine learning method with something that could learn faster

and have a better prediction. Another possible path is to combine RSCAT with other algorithms and find a better match than CUBIC or DCTCP, with newer DC-centered congestion control algorithms. Finally, one last possible path of continuing this work is by creating a brand new congestion control algorithm that uses RSCAT's learning and network visibility to have a better performance without the need to inject rules in the network or inject changes in the packet header. This could be used to mitigate the problem when dealing with mice traffic.

6.1 PUBLICATIONS

Publishing's and submissions related and originated during this thesis' research:

1. Paper published and presented at the Escola Regional de Redes de Computadores (ERRC) 2020 in Porto Alegre, RS. The title of the paper is "Controle de Congestionamento em Data Center baseado em SDN e Aprendizado de Máquina: uma Proposta Preliminar".
2. Paper accepted at IEEE Global Communications Conference (GLOBECOM) 2022 in Rio de Janeiro, Brazil. The title of the paper is "Data classification and reinforcement learning to avoid congestion on SDN-based data centers".

BIBLIOGRAPHY

ABBASLOO, Soheil et al. Cellular controlled delay tcp (c2tcp). In: IEEE. **2018 IFIP Networking Conference (IFIP Networking) and Workshops**. [S.l.], 2018. p. 118–126. Cited on page 23.

ABBASLOO, Soheil; YEN, Chen-Yu; CHAO, H Jonathan. Classic meets modern: A pragmatic learning-based congestion control for the internet. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 632–647. Cited 2 times on pages 51 and 53.

ABBASLOO, Soheil; YEN, Chen-Yu; CHAO, H Jonathan. Wanna make your tcp scheme great for cellular networks? let machines do it for you! **IEEE Journal on Selected Areas in Communications**, IEEE, v. 39, n. 1, p. 265–279, 2020. Cited 2 times on pages 51 and 53.

ABDELMONIEM, Ahmed M; BENSAOU, Brahim. Implementation and evaluation of data center congestion controller with switch assistance. **arXiv preprint arXiv:2106.14100**, 2021. Cited 3 times on pages 13, 51, and 53.

AL-FARES, Mohammad; LOUKISSAS, Alexander; VAHDAT, Amin. A scalable, commodity data center network architecture. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 38, n. 4, p. 63–74, 2008. Cited on page 24.

ALIZADEH, Mohammad et al. Data center tcp (dctcp). In: **Proceedings of the ACM SIGCOMM 2010 Conference**. [S.l.: s.n.], 2010. p. 63–74. Cited 7 times on pages 13, 24, 25, 26, 50, 53, and 62.

BABAYIGIT, Bilal; ULU, Banu. Deep learning for load balancing of sdn-based data center networks. **International Journal of Communication Systems**, Wiley Online Library, v. 34, n. 7, p. e4760, 2021. Cited 2 times on pages 51 and 53.

BAILEY, D.H. et al. The nas parallel benchmarks. **Int. J. High Perform. Comput. Appl.**, Sage Publications, Inc., USA, v. 5, n. 3, p. 63–73, sep 1991. ISSN 1094-3420. Disponível em: <https://doi.org/10.1177/109434209100500306>. Cited on page 60.

BALAKIRUTHIGA, B; DEEPALAKSHMI, P. (itmp)–intelligent traffic management prototype using reinforcement learning approach for software defined data center (sddc). **Sustainable Computing: Informatics and Systems**, Elsevier, p. 100610, 2021. Cited 2 times on pages 51 and 53.

BENSON, Theophilus; AKELLA, Aditya; MALTZ, David A. Network traffic characteristics of data centers in the wild. In: **Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement**. [S.l.: s.n.], 2010. (IMC '10), p. 267–280. ISBN 9781450304832. Cited on page 60.

BHATNAGAR, Shalabh et al. Natural actor–critic algorithms. **Automatica**, Elsevier, v. 45, n. 11, p. 2471–2482, 2009. Cited 2 times on pages 14 and 37.

BOUTABA, Raouf et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. **Journal of Internet Services and Applications**, Springer, v. 9, n. 1, p. 1–99, 2018. Cited 6 times on pages 14, 33, 34, 36, 38, and 39.

BOUZIDI, El Hocine; OUTTAGARTS, Abdelkader; LANGAR, Rami. Deep reinforcement learning application for network latency management in software defined networks. In: IEEE. **2019 IEEE Global Communications Conference (GLOBECOM)**. [S.l.], 2019. p. 1–6. Cited 3 times on pages 14, 52, and 53.

BRAKMO, Lawrence S; O'MALLEY, Sean W; PETERSON, Larry L. Tcp vegas: New techniques for congestion detection and avoidance. In: **Proceedings of the conference on Communications architectures, protocols and applications**. [S.l.: s.n.], 1994. p. 24–35. Cited on page 22.

BRAKMO, Lawrence S.; PETERSON, Larry L. Tcp vegas: End to end congestion avoidance on a global internet. **IEEE Journal on selected Areas in communications**, IEEE, v. 13, n. 8, p. 1465–1480, 1995. Cited on page 22.

CARDWELL, Neal et al. Bbr: Congestion-based congestion control. **ACM Queue**, v. 14, September-October, p. 20 – 53, 2016. Disponível em: <http://queue.acm.org/detail.cfm?id=3022184>. Cited 3 times on pages 20, 50, and 53.

CARDWELL, Neal et al. Bbr: congestion-based congestion control. **Communications of the ACM**, ACM New York, NY, USA, v. 60, n. 2, p. 58–66, 2017. Cited on page 23.

CHANG, Hyunseok et al. Magnet: Machine learning guided application-aware networking for data centers. **IEEE Transactions on Cloud Computing**, IEEE, 2021. Cited 2 times on pages 51 and 53.

CHIU, Dah-Ming; JAIN, Raj. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. **Computer Networks and ISDN systems**, Elsevier, v. 17, n. 1, p. 1–14, 1989. Cited 2 times on pages 14 and 24.

CRONKITE-RATCLIFF, Bryce et al. Virtualized congestion control. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 230–243. Cited 6 times on pages 7, 13, 14, 27, 28, and 46.

DIEL, Gustavo; KOSLOVSKI, Guilherme Piêgas. Uma proposta para gerenciamento do tráfego do protocolo dropbox lan sync em redes definidas por software. In: SBC. **Anais da XIX Escola Regional de Alto Desempenho da Região Sul**. [S.l.], 2019. Cited on page 31.

DIEL, Gustavo; MARCONDES, Anderson; KOSLOVSKI, Guilherme. Uma ferramenta para evitar enlaces congestionados em redes definidas por software. In: SBC. **Anais da XVII Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul**. [S.l.], 2017. Cited on page 31.

DUKKIPATI, Nandita; MCKEOWN, Nick. Why flow-completion time is the right metric for congestion control. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 36, n. 1, p. 59–62, 2006. Cited on page 58.

EOM, Won-Ju et al. Network traffic classification using ensemble learning in software-defined networks. In: IEEE. **2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)**. [S.l.], 2021. p. 089–092. Cited 2 times on pages 51 and 53.

ERICKSON, Bradley J et al. Machine learning for medical imaging. **Radiographics**, Radiological Society of North America, v. 37, n. 2, p. 505–515, 2017. Cited on page 34.

ESTRADA-SOLANO, Felipe; CAICEDO, Oscar M; FONSECA, Nelson LS Da. Nelly: Flow detection using incremental learning at the server side of sdn-based data centers. **IEEE Transactions on Industrial Informatics**, IEEE, v. 16, n. 2, p. 1362–1372, 2019. Cited 3 times on pages 52, 53, and 54.

FACEBOOK. **Facebook Traffic Traces**. 2017. <https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network>. Cited on page 38.

FADLULLAH, Zubair Md et al. State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems. **IEEE Communications Surveys & Tutorials**, IEEE, v. 19, n. 4, p. 2432–2455, 2017. Cited on page 36.

FLOYD, Sally et al. The newreno modification to tcp's fast recovery algorithm. RFc 2582, April, 1999. Cited 2 times on pages 13 and 22.

FOUNDATION, Open Networking. Specification, **OpenFlow Version 1.3.0**. 2012. Https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf. Cited 8 times on pages 14, 28, 29, 30, 31, 39, 46, and 55.

FRALEIGH, Chuck et al. Design and deployment of a passive monitoring infrastructure. In: SPRINGER. **Thyrrhenian Internatinal Workshop on Digital Communications**. [S.l.], 2001. p. 556–575. Cited on page 39.

FU, Qiongxiao et al. Deep q-learning for routing schemes in sdn-based data center networks. **IEEE Access**, IEEE, v. 8, p. 103491–103499, 2020. Cited 3 times on pages 14, 52, and 53.

GENTLEMAN, Robert; CAREY, Vincent J. Unsupervised machine learning. In: **Bioconductor case studies**. [S.l.]: Springer, 2008. p. 137–157. Cited 2 times on pages 33 and 34.

GOMEZ, Jose et al. A performance evaluation of tcp bbrv2 alpha. In: . [S.l.: s.n.], 2020. Cited on page 23.

GROUP, WAND Network Research. **Waikato Internet Traffic Storage**. 2013. <https://wand.net.nz/wits>. Cited on page 38.

HA, Sangtae; RHEE, Injong; XU, Lisong. Cubic: a new tcp-friendly high-speed tcp variant. **ACM SIGOPS operating systems review**, ACM New York, NY, USA, v. 42, n. 5, p. 64–74, 2008. Cited 3 times on pages 20, 22, and 62.

HANDLEY, Mark. Why the internet only just works. **BT Technology Journal**, Springer, v. 24, n. 3, p. 119–129, 2006. Cited 3 times on pages 17, 18, and 20.

HE, Keqiang et al. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.: s.n.], 2016. p. 244–257. Cited 5 times on pages 14, 27, 46, 51, and 53.

HOCK, Mario; BLESS, Roland; ZITTERBART, Martina. Experimental evaluation of bbr congestion control. In: IEEE. **2017 IEEE 25th International Conference on Network Protocols (ICNP)**. [S.l.], 2017. p. 1–10. Cited on page 23.

ISLAM, Safiqul; WELZL, Michael; GJESSING, Stein. How to control a tcp: Minimally-invasive congestion management for datacenters. In: IEEE. **2019 International Conference on Computing, Networking and Communications (ICNC)**. [S.l.], 2019. p. 121–125. Cited on page 27.

JAAKKOLA, Tommi; JORDAN, Michael I; SINGH, Satinder P. On the convergence of stochastic iterative dynamic programming algorithms. **Neural computation**, MIT Press, v. 6, n. 6, p. 1185–1201, 1994. Cited 2 times on pages 14 and 37.

JACOBSON, Van. Congestion avoidance and control. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 18, n. 4, p. 314–329, 1988. Cited 3 times on pages 19, 20, and 21.

JACOBSON, Van. Modified tcp congestion control and avoidance alogrithms. **End-2-end-interest mailing list**, 1990. Cited on page 22.

KANDULA, Srikanth et al. The nature of data center traffic: measurements & analysis. In: **Proceedings of the 9th ACM SIGCOMM conference on Internet measurement**. [S.l.: s.n.], 2009. p. 202–208. Cited 3 times on pages 13, 17, and 18.

KIM, Hyojoon; FEAMSTER, Nick. Improving network management with software defined networking. **IEEE Communications Magazine**, IEEE, v. 51, n. 2, p. 114–119, 2013. Cited on page 29.

KONDA, Vijay R; TSITSIKLIS, John N. Actor-critic algorithms. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2000. p. 1008–1014. Cited on page 37.

KREUTZ, Diego et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, Ieee, v. 103, n. 1, p. 14–76, 2014. Cited 6 times on pages 7, 28, 29, 30, 31, and 44.

KUZMANOVIC, Aleksandar et al. Adding explicit congestion notification (ecn) capability to tcp's syn/ack packets. **RFC5562**, 2009. Cited on page 25.

LA, Richard J; WALRAND, Jean; ANANTHARAM, Venkatachalam. **Issues in TCP vegas**. [S.l.]: Citeseer, 1999. Cited 2 times on pages 13 and 23.

LANTZ, Bob; HELLER, Brandon; MCKEOWN, Nick. A network in a laptop: rapid prototyping for software-defined networks. In: **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2010. p. 1–6. Cited on page 55.

MARCONDES, Anderson HS et al. Executing distributed applications on sdn-based data center: A study with nas parallel benchmark. In: IEEE. **2016 7th International Conference on the Network of the Future (NOF)**. [S.l.], 2016. p. 1–3. Cited on page 69.

Measurement Lab. **The M-Lab NDT Data Set**. 2009. <https://measurementlab.net/tests/ndt>. Bigquery table measurement-lab.ndt.download. Cited on page 38.

MENTZ, Lucas Litter; LOCH, Wilton Jaciel; KOSLOVSKI, Guilherme Piêgas. Comparative experimental analysis of docker container networking drivers. In: **IEEE Int. Conference on Cloud Networking (CloudNet)**. [S.l.: s.n.], 2020. p. 1–7. Cited on page 60.

MORO, Vilson et al. Analysis of virtualized congestion control in applications based on hadoop mapreduce. In: BIANCHINI, Calebe et al. (Ed.). **High Performance Computing Systems**. Cham: Springer International Publishing, 2020. p. 37–52. ISBN 978-3-030-41050-6. Cited on page 27.

NAGLE, John. **RFC0896: Congestion control in IP/TCP internetworks**. [S.l.]: RFC Editor, 1984. Cited 3 times on pages 17, 18, and 20.

NOORMOHAMMADPOUR, Mohammad; RAGHAVENDRA, Cauligi S. Datacenter traffic control: Understanding techniques and tradeoffs. **IEEE Communications Surveys & Tutorials**, IEEE, v. 20, n. 2, p. 1492–1525, 2017. Cited 2 times on pages 13 and 24.

NUNES, Bruno Astuto A et al. A survey of software-defined networking: Past, present, and future of programmable networks. **IEEE Communications surveys & tutorials**, IEEE, v. 16, n. 3, p. 1617–1634, 2014. Cited on page 29.

PASZKE, Adam et al. Pytorch: An imperative style, high-performance deep learning library. In: **Advances in Neural Information Processing Systems 32**. Curran Associates, Inc., 2019. p. 8024–8035. Disponível em: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>. Cited on page 56.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011. Cited 2 times on pages 56 and 62.

PRAKASH, Pawan et al. The {TCP} outcast problem: Exposing unfairness in data center networks. In: **9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)**. [S.l.: s.n.], 2012. p. 413–426. Cited 2 times on pages 24 and 25.

SCHULTZ, Wolfram; DAYAN, Peter; MONTAGUE, P Read. A neural substrate of prediction and reward. **Science**, American Association for the Advancement of Science, v. 275, n. 5306, p. 1593–1599, 1997. Cited on page 36.

STEVENS, W. Richard. **TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms**. RFC Editor, 1997. RFC 2001. (Request for Comments, 2001). Disponível em: <https://rfc-editor.org/rfc/rfc2001.txt>. Cited 4 times on pages 17, 18, 20, and 21.

STOICA, I. **A Comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas**. [S.l.]: University of California Berkeley, CA, 2005. Cited 3 times on pages 13, 21, and 22.

SURYAVANSHI, Ajay Kumar Mahendra N. Tcp incast in data center networks: Issue and existing solutions. In: **International Journal of Innovative Technology and Exploring Engineering (IJITEE)**. [S.l.: s.n.], 2019. Cited 2 times on pages 13 and 26.

SUTTON, Richard S. Learning to predict by the methods of temporal differences. **Machine learning**, Springer, v. 3, n. 1, p. 9–44, 1988. Cited on page 36.

SUTTON, Richard S et al. Policy gradient methods for reinforcement learning with function approximation. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2000. p. 1057–1063. Cited on page 36.

TALPUR, Ali. **Congestion Detection in Software Defined Networks using Machine Learning**. Tese (Doutorado) — PhD thesis, 02 2017, 2017. Cited 2 times on pages 52 and 53.

TESAURO, Gerald. Practical issues in temporal difference learning. **Machine learning**, Springer, v. 8, n. 3, p. 257–277, 1992. Cited on page 36.

TESAURO, Gerald. Reinforcement learning in autonomic computing: A manifesto and case studies. **IEEE Internet Computing**, IEEE, v. 11, n. 1, p. 22–30, 2007. Cited on page 36.

TESAURO, Gerald et al. Temporal difference learning and td-gammon. **Communications of the ACM**, v. 38, n. 3, p. 58–68, 1995. Cited on page 36.

TOSOUNIDIS, Vasileios; PAVLIDIS, Georgios; SAKELLARIOU, Ilias. Deep q-learning for load balancing traffic in sdn networks. In: **11th Hellenic Conference on Artificial Intelligence**. [S.l.: s.n.], 2020. p. 135–143. Cited 2 times on pages 51 and 53.

WU, Haitao et al. Ictcp: Incast congestion control for tcp in data-center networks. **IEEE/ACM transactions on networking**, IEEE, v. 21, n. 2, p. 345–358, 2012. Cited on page 24.

WU, Haitao et al. Tuning ecn for data center networks. In: **Proceedings of the 8th international conference on Emerging networking experiments and technologies**. [S.l.: s.n.], 2012. p. 25–36. Cited on page 25.

ZHANG, Lin Cui Yuxiang et al. Mystique: A fine-grained and transparent congestion control enforcement scheme. 2019. Cited 3 times on pages 13, 50, and 53.

ZHU, Xiaojin; GOLDBERG, Andrew B. Introduction to semi-supervised learning. **Synthesis lectures on artificial intelligence and machine learning**, Morgan & Claypool Publishers, v. 3, n. 1, p. 1–130, 2009. Cited on page 34.