



POLITECNICO  
DI TORINO

POLITECNICO DI TORINO

Master Degree Thesis

# Delay Control with Programmable Data Planes

## Relatori

Prof. Paolo GIACCONE

Prof. Andrea BIANCO

Farhad FATHI

ACADEMIC YEAR 2023-2024

# Acknowledgements

I would like to express my deepest gratitude to my supervisors, Prof. Giaccone and Prof. Bianco, for their invaluable guidance, expertise, and continuous support throughout the entire research process. Their insightful feedback, encouragement, and mentorship have been instrumental in shaping the direction and quality of this thesis. I am truly grateful for their dedication and commitment to my academic growth.

I am also indebted to Alessandro Cornacchia, a Ph.D. student, for his exceptional assistance and collaboration during this research project. His technical expertise, critical insights, and willingness to engage in detailed discussions have greatly enriched my understanding of the subject matter. His contributions have been instrumental in the success of this work.

I would like to extend my sincere appreciation to the Department of Communication and Computer Network Engineering at Politecnico di Torino for providing a stimulating academic environment and access to valuable resources. The support and facilities offered by the department have played a significant role in the successful completion of this thesis.

Finally, I would like to thank all the individuals who have played a role, however small, in the completion of this thesis. Your contributions, assistance, and encouragement are sincerely appreciated.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on “Telecommunications of the Future” (PE000000001 - program “RESTART”)

## Abstract

This thesis looks into the topic of "Delay Control with Programmable Data Planes". The primary focus is on exploring the potential of programmable switches, specifically BMv2 switches programmed with the P4 language. By investigating the operations that can be performed inside switch pipelines, the thesis aims to provide a deeper understanding of how to control the delay of packets in the network.

Through our attempts, we observed that recirculation within the programmable switch is a potential method for controlling packet delays across the network. In the following, I developed packet recirculation within programmable switches. Based on the development of packet recirculation, this study investigates the effect of different recirculation iterations on packet delay. Additionally, we explore the broader consequences of this packet recirculation approach on network throughput. Initially, we conducted experiments in a controlled virtual environment. To enhance the depth of our insights, we extend our experimentation to real scenarios, employing a Linux server. This dual approach enables us to figure out the variations in outcomes between experiments in virtual and real Linux environments.

The research employs Mininet, a network emulator, within the virtual Linux environment (Ubuntu) and the real Linux server to emulate a simple network topology. The topology consists of a source host, a destination host, and one or a chain of BMv2 switches. Programmable switches offer flexibility and programmability in their data plane, which consists of forwarding tables, flow tables, and packet processing capabilities. These switches enable customized forwarding decisions based on defined rules and match fields such as addresses, ports, or protocol types. The control plane, managed by a controller, plays a vital role in configuring and orchestrating the behavior of the programmable switches. By leveraging the flexibility and programmability of these switches, we can manage forwarding decisions or do other actions, adapt to dynamic network conditions, and achieve our goal.

To evaluate the delay of packets and the performance of the network, experiments are conducted considering variations in the link parameters and configurations. For changing link parameters, a traffic controller tool within the Mininet is utilized. Throughput measurements are performed using the widely used IPERF tool, which supports both UDP and TCP protocols. Network delay is measured using the ping utility, which calculates round-trip times between the source and a specified destination. These measurements provide valuable insights into the effects of packet recirculation on the delay of packets in the network.

The findings from this research contribute to enhancing the understanding of data plane programmability, how we can use recirculation in it to control the delay of packets, and the impacts of this on network performance, specifically throughput.

Keywords: Data Plane Programmability, Programmable Switches, BMv2, P4 Language, Throughput, Delay, Packet Recirculation, Mininet.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Background and context . . . . .	4
1.2	Motivation for Controlling Delay . . . . .	8
<b>2</b>	<b>Programmable Data Planes</b>	<b>11</b>
2.1	Programmable packet processing pipelines . . . . .	13
2.2	Programming languages and compilers . . . . .	14
<b>3</b>	<b>Related work</b>	<b>19</b>
<b>4</b>	<b>Delay Control by Packet Recirculation</b>	<b>37</b>
4.1	Packet Recirculation development . . . . .	38
4.2	How to Control Delay? . . . . .	52
<b>5</b>	<b>Experimental/numerical evaluation</b>	<b>57</b>
5.1	Methodology . . . . .	57
5.2	Numerical results . . . . .	59
<b>6</b>	<b>Conclusion</b>	<b>83</b>
	<b>Bibliography</b>	<b>85</b>

# Chapter 1

## Introduction

The evolution of next-generation internet technologies, such as 5G and the future 6G, has resulted in an increasing number of use cases for Software-Defined Networking (SDN). This novel networking paradigm contrasts with the conventional one, which depends on specific hardware devices such as routers and switches to regulate network traffic. SDN, on the other hand, presents a software-based method that enables centralized control and programmability over these hardware devices through controllers. The separation of the control plane and the forwarding plane enables more efficient administration, configuration, and orchestration of network resources, liberating us from the confines of outdated techniques and protocols. SDN provides a flexible and adaptable infrastructure for routing network traffic and implementing network policies. Network administrators may flexibly adjust to changing requirements, divert traffic flows, and improve the utilization of resources by separating the control and forwarding operations. Because of its programmability, administrators may specify network behavior depending on application requirements, resulting in more efficient and effective network operations.

One of the key devices in SDN that plays a critical role in facilitating communication in the network is the programmable switch. Programmable switches are the main focus of this research, as they offer the capability to customize their behavior and functions through programmability. This research aims to investigate the functions and behavior of programmable switches and understand their impact on the network. By examining how programmable switches operate within SDN, valuable insights can be gained in delay control of the packets in the network.

For investigating the impacts of the behavior and functions of programmable switches, it is crucial to have an environment where experiments can be conducted. In this research, a Linux virtual environment was utilized to simulate the experiments. Mininet, a powerful network emulator, proved to be a valuable tool for conducting these simulations. It allows for the creation of various network topologies and facilitates the emulation of programmable switches. One such important component used in the experiments is the BMv2 switch. This software switch accurately simulates the behavior of programmable switches, enabling a realistic evaluation of their functions and impact on the network.

To evaluate the effects of programmable switch behavior on network performance, it is essential to define the concept of performance itself. Two critical parameters that

provide valuable insights into network performance are delay and throughput. Delay refers to the time it takes for a packet to travel from the source to the destination, while throughput measures the amount of data that can be transmitted over a network within a given timeframe. These parameters serve as key indicators for assessing the efficiency and effectiveness of network communication.

One of the key aspects that this research focuses on is the recirculation of packets in programmable switches. Recirculation is a technique employed in P4 switches to handle packets that require additional processing, such as those needing more than one lookup or complex processing functions.

Recirculation serves various purposes, including managing latency, enhancing throughput, and implementing advanced processing functions. In a recirculation-based architecture, the switch redirects packets back to the ingress pipeline for further processing, introducing the potential for additional latency and delay.

The behavior of recirculation in P4 switches is highly configurable. The P4 language allows defining conditions under which a packet should be recirculated. Moreover, the P4 switch can determine how packets should be forwarded once the recirculation period concludes.

To measure delay and throughput, appropriate tools are required. In this research, two simple yet powerful tools available within the Mininet network emulator are employed. For measuring delay, the ping utility is utilized. To measure throughput, the widely used tool Iperf is employed.

By exploring the behavior and implications of recirculation in programmable switches and collecting measurements and observations, this research aims to understand how, with this ability, we can control the delay of packets in the network. Investigating recirculation also enables valuable insights into how this technique impacts the overall throughput of the network.

## 1.1 Background and context

### 1.1.1 The evolution of networks

In recent times, the world of networking technology has experienced an exhilarating wave of advancements that have fundamentally transformed how we connect and communicate in the digital age. These innovations have been like a fast-forward button, reshaping the very foundations of networking and how we interact with the virtual world around us.

The speed at which these innovations have unfolded is truly remarkable. As our emphasis on digital interactions has expanded exponentially, networking technologies have risen to the challenge, pushing the boundaries of what we thought was possible. This rapid evolution has triggered a chain reaction of game-changing developments, each contributing to the dynamic landscape of modern communication.

One of the standout achievements has been in the field of high-speed data transmission. Think about fiber-optic networks, which leverage the speed of light to transmit data at terrifying rates. This has completely transformed how we interact with digital content, making seamless streaming, real-time collaboration, and instant sharing of multimedia an everyday reality.

Wireless networking hasn't been left behind either. The shift from traditional Wi-Fi to the power of 5G cellular technology has been a game-changer. With its remarkably low latency and unparalleled bandwidth, 5G has opened up a world of possibilities – not just for smartphones and laptops but for IoT, smart cities, self-driving cars, and even mind-bending augmented reality experiences.

Then there's the fascinating world of software-defined networking (SDN), which has revolutionized how we manage and control networks. By detaching network control from physical hardware, SDN offers a programmable framework that adapts to changing needs, streamlining operations and giving a serious boost to security.

These advances, taken together, paint a vivid picture of a networking world that's moved far beyond just connecting devices. It's a world of lightning-fast speeds, intelligent networks, and unmatched flexibility. As we go into the complexities of controlling delays in programmable data planes, it's essential to understand the context of this rapid shift. This thesis goes deeply into the core of these advancements, investigating their influence and maximizing the potential of programmable data planes, all while shaping the very future of networking as we know it.

## Traditional Networks

In the traditional network approach, devices like routers and switches held dual responsibilities: making decisions about how data should traverse the network and physically forwarding the data. Each device operated based on predefined rules and protocols, communicating with neighboring devices to guide traffic. While this approach was functional, it posed challenges when adapting to changing demands, scaling, and introducing new services. Manual configuration of each device led to complexity, and network-wide changes were cumbersome.

## SDN

SDN redefined networking as a group of members led by a core component. In this paradigm, the control plane, responsible for decision-making, was abstracted and centralized within an SDN controller. Devices in the network -the data plane- followed the instructions of the controller. This decoupling allowed for dynamic control of traffic flows, enabling rapid adjustments to changing conditions. The SDN controller became the heart of network intelligence, managing data movement in response to real-time demands.

***Why the Shift to SDN?*** The transition to SDN was driven by a convincing need for networks to keep pace with rapidly evolving technology and user demands. Traditional networks struggled to accommodate the exponential growth of data, the rise of cloud computing, and the emergence of new applications. SDN emerged as a solution to these challenges, offering unprecedented agility, adaptability, and efficiency. By separating control from data forwarding, SDN empowers network administrators to dynamically shape traffic, respond to security threats, and optimize performance. It simplifies management, accelerates innovation, and positions networks to embrace future technological advancements. In essence, SDN represents a leap from a scripted performance to an orchestrated symphony, where the conductor – the SDN controller – harmonizes the

complex interplay of data in a way that the traditional approach could not achieve. This transformation isn't just about network technology; it's a paradigm shift that empowers networks to progress in the dynamic digital landscape.

### 1.1.2 Network performance

Network performance is a key characteristic of modern communication systems, including a variety of factors that collectively characterize the efficiency, dependability, and responsiveness of data transfer inside a network. Understanding network performance and its related characteristics is crucial for improving data delivery, providing smooth user experiences, and ensuring critical applications run efficiently. In this part, we will look at the important factors that influence network performance.

#### Latency

Latency, often referred to as network delay, represents the time it takes for a packet of data to travel from its source to its destination within a network. It is a critical metric in network performance, influencing real-time applications such as video conferencing, online gaming, and financial trading. Latency is typically categorized into several components:

**Propagation Delay:** The time taken for data to traverse the physical distance between sender and receiver. In practice, this delay is affected by the speed of light and the medium through which data travels (e.g., fiber optic cable or copper wire). Propagation delay is largely determined by the physical characteristics of the network.

**Transmission Delay:** The time required to push all the packet's bits into the communication channel. It is inversely proportional to the bandwidth of the link and directly related to the size of the packet. Higher bandwidth and smaller packet sizes result in reduced transmission delay.

**Processing Delay:** The time taken by devices within the network (routers, switches, hosts) to process and forward the packet. This includes tasks such as routing table lookups, error checking, etc. Minimizing processing delay is essential for efficient data transmission.

**Queuing Delay:** The delay incurred when packets wait in queues at network devices, such as routers, before being transmitted. Queuing delay is affected by network congestion, packet prioritization, and the overall load on network equipment.

Minimizing latency is critical in scenarios where real-time responses are essential, such as voice and video communication or autonomous vehicles. High-latency networks can lead to sluggish user experiences, dropped calls, and missed opportunities.

#### Throughput

Throughput measures the rate at which data is successfully transmitted from source to destination within a network. It is often quantified in bits per second (bps) and reflects the network's capacity to handle data traffic. Throughput is affected by various factors, including network bandwidth, packet loss, and network congestion.



**Bandwidth:** Network bandwidth represents the maximum data transfer rate a network link can support. It is a crucial determinant of network throughput. Higher bandwidth enables the transmission of larger volumes of data in a shorter time.

**Packet Loss:** Packet loss occurs when data packets do not reach their destination due to network congestion or errors. High packet loss rates can significantly degrade throughput and result in data retransmissions.

**Network Congestion:** Congestion arises when network resources are insufficient to accommodate the volume of data traffic. It leads to increased packet queuing and delays, which can impact throughput. Effective congestion management is essential for maintaining high throughput.

Optimizing throughput is essential for applications involving large data transfers, such as file downloads, video streaming, and cloud-based services. A well-performing network should balance low latency with high throughput to deliver a seamless and responsive user experience.

## Reliability

Reliability in network performance refers to the network's ability to consistently deliver data without errors or interruptions. Reliable networks minimize data loss, maintain low latency, and ensure the consistent availability of network resources. Achieving network reliability often involves redundancy, fault tolerance mechanisms, and effective error correction protocols.

## Jitter

Jitter represents the variation in latency or delay between data packets in a network. In real-time applications like VoIP (Voice over Internet Protocol) and video conferencing, consistent and low jitter is crucial. High jitter can result in disruptions and poor quality in audio and video streams.

Understanding these key parameters of network performance is fundamental for optimizing network design, troubleshooting issues, and ensuring the efficient operation of modern communication systems. In the following sections, we explore how programmable switches and packet recirculation in them can impact these critical aspects of network performance.

As we go into the core of influencing network performance and controlling delays inside programmable data planes, it's critical to remember that we're not simply looking at technical details. We're getting to the heart of what it means to live in a connected world, where the quality of our relationships, the success of our companies, and the growth of our society all rely on these fundamental concepts.

### 1.1.3 The emergence of programmable switches as a key player in modern network infrastructure

Programmable switches act like advanced controllers in today's fast-paced digital environment, constantly controlling data flows and improving network operations. They

can change and reconfigure instantly, ensuring efficient data transfer and smooth connection between devices. Because of this flexibility, programmable switches are an essential component of modern network architecture.

Think of them as digital traffic controllers. They help data flow efficiently, reducing delays and bottlenecks. Imagine a highway with no traffic jams –that’s what these switches aim to create in the digital world.

In fact, these switches are key players in modern networks. They’re like the brains behind the scenes, making sure data gets where it needs to go without getting lost or taking too long. This is crucial in today’s world, where we need things to happen quickly, from streaming videos to sending important files.

As we go deeper into this thesis, we’ll explore how these switches work, ensuring that our networks run smoothly, efficiently, and without any problems.

## 1.2 Motivation for Controlling Delay

### 1.2.1 Packet Delay Control by Recirculation within Programmable Switches

The concept of controlling packet delay arose from a fundamental curiosity: Can the natural sequence of packet receiving inside a network be altered? This thought led us to imagine scenarios in which packet sequence dynamics may be modified to allow a later packet to arrive before an earlier one. This imaginative exploration raised the crucial question: Could programmable switches be the key to directing such temporal reorganizations? The idea of using programmable switches to influence packet delay developed as a fascinating challenge to this desire. The initial flame was sparked by the contrast of packet arrival timings at a destination. The theory was simple: if packet A arrived at the destination before packet B, could we reverse the sequence using programmable switches to allow packet B to arrive first? With this idea, we set out on a quest to discover a mechanism that might actually modify packet flow patterns. The essence of the problem was a wish to temporarily block the flow of specific packets within programmable switches. Could we develop a method to keep a packet within a switch while allowing other packets to pass through unhindered? The desired solution became an idea of recirculation which is a procedure in which packets are actively looped back into the switch for a certain period of time. This stop and redirection approach creates an opening for controlling the precise exit time of packets, allowing for a reorganizing of the sequential delivery sequence. The goal of controlling packet delay by recirculation arose from the creative convergence of curiosity and engineering knowledge. It was the result of imagining a new approach to network management, one that goes beyond existing paradigms to bring in a new era of dynamic, performed data mobility. The core of this drive characterized every step of our study as we moved from idea to implementation, pushing us to dig into the worlds of programmable switches, packet recirculation, and various aspects of temporal control inside network topology.

### 1.2.2 Analyzing the impact of Packet Recirculation within Programmable Switches on network performance

As previously explained, the central motivation driving this thesis is the exploration of packet delay control through recirculation using programmable switches within the network. Furthermore, this investigation extends to comprehending the consequential effects of such mechanisms on network performance, with a specific focus on throughput and latency.

The importance of efficient network performance and latency in today’s digital landscape is crucial. The research aims to understand the dynamics of packet recirculation within programmable switches, a fundamental concept in network design, and its impact on network performance and latency. The study aims to uncover insights that can inform strategies for controlling delay and improving data transmission efficiency. Through rigorous analysis and experimentation, the thesis contributes to the broader understanding of network performance management in the context of programmable switches. The implications of packet recirculation can shape the design, optimization, and utilization of modern network infrastructures, ultimately facilitating the development of more efficient and responsive networks to address the demands of today’s data-intensive and interconnected world.



## Chapter 2

# Programmable Data Planes

Computer networks are the cornerstone of modern technical infrastructures, but because of their widespread use and variability, it is more difficult to design network systems and the components that make them up. The demand for programmable networking hardware that enables operators to modify device functionality through a programming interface is a result of the conflict between specialization and making network devices commercialized and universal. Network operators are now able to quickly add new features without having to wait for long release cycles because of the change in the interaction between device suppliers and network operators. Additionally, programmability enables hardware manufacturers to devote technical resources to refining a set of clearly defined building blocks for customized logic. Operators who have to support enormous machine learning, big data applications, large-scale cloud computing, and the 5G mobile standard will find this new generation of programmable devices to be of special assistance. For these applications to handle constantly changing and diverse sets of protocols and services, network hardware such as switches, middleboxes, and network interface cards (NICs) are needed. The design, production, testing, and deployment of specialized hardware components are all necessary for traditional fixed-function devices and are time- and money-consuming engineering tasks. Rising new functionality is slow and expensive, forcing providers to only implement features when they are heavily desired, which restricts innovation. It is inefficient to include each and every network protocol in a device's packet processing logic since it uses up valuable memory space and CPU time for capabilities that very few operators will ever use. These problems are addressed by the advent of programmable network devices, which enable thorough reconfiguration of packet processing capability. Devices in both software and hardware should be programmable. The vast array of processing primitives offered by new software-based network switches, which operate on general-purpose CPUs, enable reconfigurability and allow pipelines to be created using conventional programming approaches. On a single commodity server, these switches can provide forwarding throughput in the order of tens of Gbit/s. A domain-specific language or a dialect of a general-purpose language can be used to combine numerous low-level primitives into complicated network operations in programmable hardware components and devices like programmable NICs and programmable switches. Many questions are still unsolved despite programmable data plane technology's increasing acceptance and

appeal. To support the widest range of network applications at the highest performance, how can we adapt and use fundamental packet processing primitives, expose the complicated processing logic to the operator for simple, safe, and verifiable configuration, and abstract, replicate, and monitor the transient packet processing states deeply ingrained in this logic? These issues are now some of the ones that the networking community is debating the most. The device control plane and device data plane are the functional divisions found in common network hardware like switches and routers. The device control plane is in charge of setting packet processing policies and maintaining the device, whereas the device data plane executes these regulations, frequently with strict performance constraints. Through distributed routing protocols, the control planes of various devices within a network context communicate with one another. The network control plane has evolved as a conceptually centralized controller with the advent of the Software-defined Networking (SDN) paradigm, with some device control plane tasks being split off and relocated to this network-level functionality. The network control plane keeps track of the data plane's devices, accepts high-level, network-wide policies via a northbound controller interface, assembles these intents into per-device packet processing policies, and then applies these policies to specific devices via a southbound controller interface. Individual switches in this design receive preset packet forwarding rules from the network control plane rather than having to build the logic necessary to maintain these policies locally. Standardized southbound APIs and protocols, such as OpenFlow, ForCES, and P4Runtime, are used for controller-switch communication. The device control plane still manages the device data plane and ends control links to the distant network control plane inside the SDN architecture, therefore it does not entirely disappear. (see Figure 2.1)

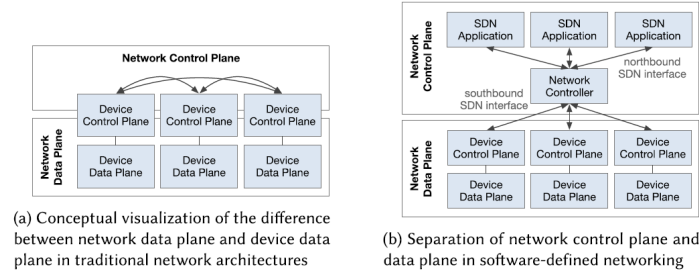


Figure 2.1. Traditional vs. SDN-based network architectures. Reproduced from [1]

Device functionality has been transformed, becoming more adaptable and dynamic thanks to the SDN paradigm. Data plane functionality in typical network equipment is tightly integrated into hardware and software, making it challenging to alter over the device's lifespan. To modify the functionality of the data plane, however, software-based packet processing systems need significant vendor changes. Almost all data plane activities are impacted by this fixed functionality, which also includes fixed entry format

and semantics, fewer protocol headers and fields, established queuing rules, and monitoring data. Data plane devices may now be fully or partially modified from the network control plane thanks to SDN and more generic hardware designs. As a result, the term "programmable data plane" came to be to describe the new kind of network hardware that enables dynamic, programmatic changes to the fundamental functioning of packet processing. A network device's ability to expose low-level packet processing logic to the control plane via a defined API is referred to as data plane programmability. This enables systematic, quick, and thorough reconfiguration. While data plane programmability was first aimed primarily at switches (particularly in data centers), a broader range of devices and functions now support low-level programmability. Programmable data plane hardware or software is increasingly utilized for general network processing and middleboxes (e.g., firewalls or load balancers) in addition to packet switching. At the edge of the network, programmable NICs allow data plane programming. These devices are available in a variety of architectures, including ASICs, FPGAs, and network processors. Due to specific hardware elements like Ternary Content Addressable Memory chips (TCAM) for effective packet matching, hardware platforms have excellent performance. Software data plane devices use improved algorithms and data structures to carry out processing logic on a common CPU.

## 2.1 Programmable packet processing pipelines

The primary function of programmable data planes is flexible packet processing, and modern packet processing pipelines are based on two essential abstractions: the match-action pipeline abstraction and the data flow graph abstraction [1]. Early designs for programmable switches employed data flow graphs, which describe processing logic as a graph with nodes denoting fundamental computation steps and edges denoting data flow from one computation stage to the next. Because of this simplicity, a programmer may easily put together useful programs utilizing a pre-existing cognitive representation of a graph of processing nodes (see Figure 2.2). A set of lookup tables arranged in a

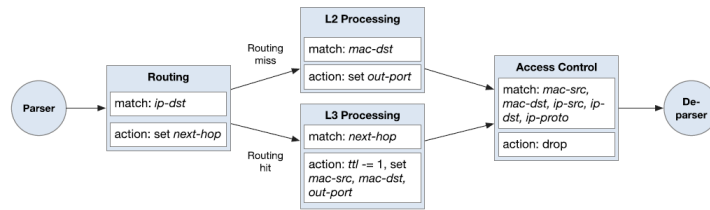


Figure 2.2. Simplified match-action table dependency graph for a basic router. Reproduced from [1]

hierarchical form is used to represent match-action processing. The switch can be told to rewrite packet contents, encapsulate/decapsulate tunnel headers, discard or forward the packet, or delay packet processing to later flow tables by using a subset of packet header

fields to do a table lookup. Using a defined API, the programmer configures the flow tables' content to dynamically set the behavior of packet processing (see Figure 2.3). The

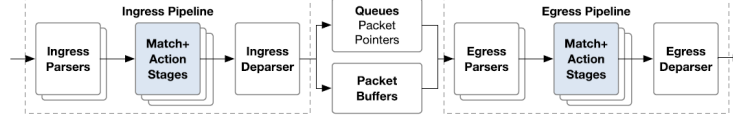


Figure 2.3. The architecture of an RMT-like switch. Reproduced from [1]

OpenFlow protocol, which permitted the development of just one flow table using a constrained number of header fields, popularized the match-action concept for programming switches. The Reconfigurable Match Tables (RMT) abstraction expands the previously constrained range of packet processing actions accessible and enables match-action tables to be created on any header field, thereby overcoming the primary constraints of OpenFlow ASICs.

## 2.2 Programming languages and compilers

High-level data plane programming languages that enable the specification of packet processing strategies inside a particular switch architecture have gained popularity during the previous several years. The requirements of operators for more complicated SDN applications and the capabilities of contemporary, more adaptable, and programmable line rate networking hardware are what motivate the development of these languages. The most well-known high-level data plane programming languages are Pyretic, Net-Core, Maple, and NetKAT. The dilemma of how to describe and change the low-level architecture and configuration of programmable switching circuits in an expressive and flexible manner is at the heart of today's programmable data planes. P4 was the first and most widely used language abstraction and compiler for expressing low-level packet processing capability within programmable data planes. Motivated by the limitations of existing SDN control protocols such as OpenFlow, which only allow for a limited set of header fields and actions, P4 allows for the definition of packet processing pipelines that include parsers and deparsers, match-action tables, and low-level operations that are applied to each packet. By matching on arbitrary bit ranges and performing user-defined actions, this language abstraction enables protocol-independent packet processing. These abstract P4 applications are built for a certain underlying data plane target. The built data plane application is then used to set up the underlying hardware or software target, and the P4-defined match-action tables are populated at runtime using a control interface like P4Runtime. P4 quickly achieved widespread acceptance in the scientific community and is now employed in a wide range of projects. P4 is a major enabling technology for extensive and flexible data plane programmability due to the large variety of supported targets, which vary from software switches to fully reconfigurable ASICs, as well as widespread industry usage. In Figure 2.4, you can see the comparison of languages



and protocols used in programmable data planes.

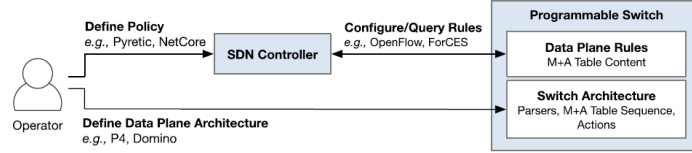


Figure 2.4. Comparison of languages and protocols used in programmable data planes. Reproduced from [1]

### 2.2.1 P4

The domain-specific programming language known as Programming Protocol-Independent Packet Processors (P4) serves the purpose of enabling the programming of packet forwarding. P4’s primary design objectives encompass three key aspects [2]. Firstly, it strives for target independence, ensuring that P4 programs can be compiled for diverse hardware types, including general-purpose CPUs, FPGAs, system(s)-on-chip, network processors, and ASICs. Each of these targets necessitates a compiler that transforms the P4 source code into a compatible switch model. Secondly, P4 emphasizes protocol independence, meaning that it lacks inherent support for common protocols like IP, Ethernet, and TCP. Instead, P4 programmers describe the necessary protocol header formats and field names within the program itself, which are then interpreted and processed by the compiled program and target device. Lastly, P4 fosters reconfigurability, enabled by its abstract language model and protocol independence. P4 targets possess the capability to modify their packet processing methods, even post-deployment, a feature traditionally linked with general-purpose CPUs and network processors, but now extended to fixed-function ASICs.

Both OpenFlow and P4 play vital roles within the realm of Software Defined Networking (SDN), where the separation of the control plane from the forwarding plane grants network operators programmatic control (see Figure 2.5).

OpenFlow, as a widely adopted interface, facilitates the control plane’s management of various forwarding devices across hardware and software vendors. Initially, the OpenFlow interface featured a straightforward rule table abstraction that matched packets based on a limited set of header fields. However, the specification has since grown in complexity to accommodate numerous header fields and rule table stages, aiming to expose switches’ enhanced capabilities to controllers. Proposing a different approach, future switches should integrate versatile mechanisms for packet parsing and header field matching, accessible through a unified, open interface. P4 represents an evolution of OpenFlow’s concepts, introducing the ability to define custom headers and tables, as well as enabling explicit programming of switching logic’s control flow.

At the heart of P4 lies the fundamental concept of match-action pipelines as you can see in Figure 2.6. Conceptually, packet or frame forwarding involves a sequence of

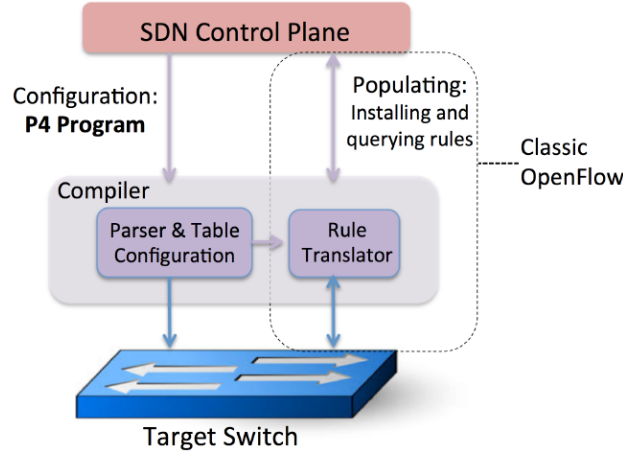


Figure 2.5. P4 for configuring switches. Reproduced from [2]

table lookups and associated header manipulations, referred to as actions in P4. Actions involve tasks such as copying byte fields based on lookup results derived from learned forwarding states. Notably, P4 focuses solely on the data plane of packet forwarding devices, leaving out specifications for the control plane and communication protocols between the control and data planes. P4's approach involves tables to represent forwarding plane states, necessitating an interface between the control plane and various P4 tables for state injection and modification.

### Components of the P4 Language

A P4 program consists of essential building blocks that collectively define its functionality:

- **Headers:** Header definitions outline packet structures and assign names to packet fields. These headers can be customized with field names and arbitrary lengths. For instance, an Ethernet header definition might include fields like "dest" and "src" each spanning 48 bits, followed by a 16-bit "type" field. These names are subsequently used within the P4 program to reference these fields.
- **Parsers:** The P4 parser, operating as a finite state machine, navigates incoming byte streams and extracts headers based on a pre-programmed parse graph. For example, it could extract Ethernet source and destination fields and further process based on the type field value (e.g., ipv4, ipv6, MPLS).
- **Tables:** P4 tables hold the state necessary for packet forwarding. Comprising lookup keys and corresponding actions with their parameters, tables serve various purposes. An example involves storing destination MAC addresses as lookup keys,

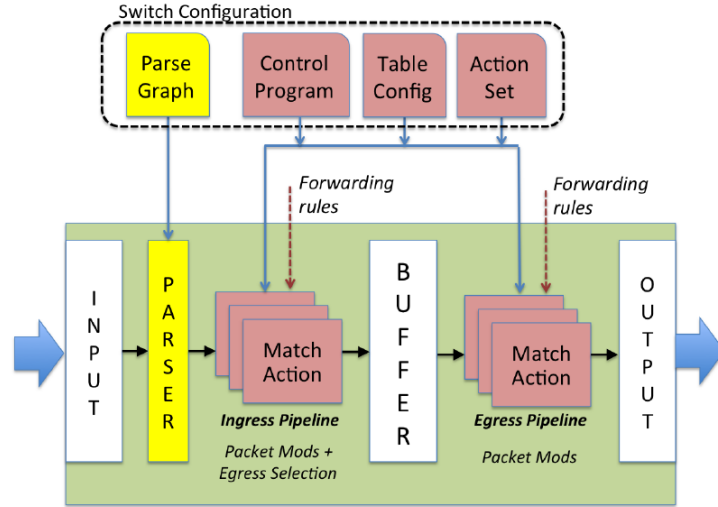


Figure 2.6. match-action pipelines for a programmable switch that is configured with P4 language. Reproduced from [2]

while the related action sets output ports or increments counters. Tables and their linked actions are typically arranged in a sequence, forming the complete packet forwarding logic, although a single comprehensive table is feasible in theory.

- **Actions:** Actions in P4 define manipulations of packet fields and metadata. Here, metadata represents packet information not directly derived from parsing, like the input interface of the incoming frame. A sample action might entail "decreasing IPv4 TTL by one" or "copying MAC address from output port table to outgoing packet header." P4 distinguishes between standard metadata applicable to all targets and target-specific metadata tailored by specific target authors.
- **Control Flow:** The control flow in P4 orchestrates table sequencing and facilitates conditional table execution through if/then/else constructs.

### Behavioral Model Framework:

P4 introduces a software reference implementation known as the behavioral model, initially written in C and generating C code based on P4 program logic. However, due to slow and complex code generation, a newer version named BMv2 has emerged in C++. A noteworthy feature of BMv2 is its decoupling of the switch's C++ code from the P4 program, which is dynamically loaded at runtime. This enables the switch to be compiled just once and permits runtime swapping of P4 programs. The BMv2 repository is designed modularly, promoting the creation of diverse targets with unique features. The primary objective of BMv2 is to facilitate hardware P4 switch vendors in modeling their

targets and reproducing behavior with different P4 programs. BMv2 switches, or Behavioral Model version 2 switches, represent a groundbreaking advancement in the realm of computer networking. These switches embody a programmable framework that fundamentally alters how network data is handled, offering a versatile canvas for configuring and orchestrating data flows. BMv2 switches play a pivotal role in the Software-Defined Networking (SDN) landscape, enabling network engineers to exercise precise control over packet processing behavior. At the core of BMv2 switches lies their dynamic packet processing capability. Think of these switches as intricate decision-making nodes within a network. When a data packet traverses a network, a BMv2 switch meticulously examines its attributes like a discerning inspector scrutinizing a package's label. Based on predetermined instructions, the switch then orchestrates how the packet should be treated—whether it should be forwarded to a specific destination, subjected to specialized processing, or even discarded. One of the defining features of BMv2 switches is their unparalleled programmability. This entails encoding a set of customized instructions that dictate the switch's response to different types of packets. Imagine a versatile maestro directing an orchestra with unique cues for each instrument—similarly, BMv2 switches are programmed to execute distinct actions based on packet characteristics. BMv2 switches prove invaluable for experimentation within a controlled environment. They facilitate the emulation of network behaviors and interactions, analogous to a virtual sandbox where engineers can build, modify, and assess intricate network scenarios without disrupting actual data traffic. BMv2 switches often operate in tandem with the P4 language—a specialized code that communicates directives to the switch, much like a conductor guiding musicians with distinct cues. This synergy between the P4 language and BMv2 switches allows for fine-tuning how packets are processed and routed, fostering adaptability in the face of evolving network demands.

## Chapter 3

# Related work

In [2], the authors provide a comprehensive exploration of the P4 language and its workings. The paper begins by elucidating the primary objectives behind the development of P4, focusing on three main goals: reconfigurability, protocol independence for switches, and the independence of packet processing functions from the underlying hardware.

To set the context, the paper delves into the history of data plane programmability and SDN, outlining their evolution and the emerging need for a language like P4. It then proceeds to explain the fundamental concept of programming hardware using P4.

The paper introduces the P4 language by drawing a comparison with the OpenFlow protocol, highlighting the distinctions between the two in various aspects. It then presents a model for packet forwarding, which is governed by two critical operations: 'configure' (programming the parser and specifying header fields for each processing stage) and 'populate' (defining how the switch should process packets).

The authors emphasize the importance of expressing dependencies between header fields, as these determine which tables can execute in parallel. They introduce the concept of Table Dependency Graphs (TDGs) for analyzing these dependencies, leading to a proposed two-step compilation process. At the highest level, imperative programming languages (P4) that capture the control flow are used to express packet processing programs. Below this level, a compiler converts the P4 representation to TDGs to aid in dependency analysis before mapping the TDG to a specific switch target. You can see the TDG for L2/L3 switch in Figure 3.1

The paper proceeds to provide a detailed explanation of P4 through an example known as the 'mTag example,' which focuses on MPLS. This example serves to illustrate essential P4 concepts such as headers, parsers, state machines, tables, match-actions, and their specifications.

The control part of P4, responsible for specifying the flow of control from one table to the next, is also elucidated, with a flowchart (Figure 3.2) and pseudocode (Figure 3.3) provided for clarity for the mTag example.

The paper sheds light on how P4 programs are compiled, mapping target-independent descriptions to specific switch hardware or software platforms. It covers the compilation process for parsers and control programs.

Additionally, the paper addresses the implementation of P4 programs across different

types of switches, including software and hardware switches. It underscores the versatility of P4 in adapting to various switch platforms.

In summary, this paper serves as a foundational resource for comprehending the P4 language and its crucial concepts. The paper's use of a well-known protocol, MPLS, in its explanations makes it an invaluable reference for us who want to use the P4 language to develop functions and new features in programmable data planes step by step using the guide of this paper.

In [3], the authors offer an extensive examination of P4 and its role within Software Defined Networks (SDN). The paper begins by tracing the evolution of networks from traditional to programmable data planes, elucidating the fundamental concepts of programmable switches and the transformation of networks from legacy to programmable models.

The paper takes a P4-centric perspective on SDN and references over 75 related research papers. It introduces the motivations for adopting SDN, primarily driven by the

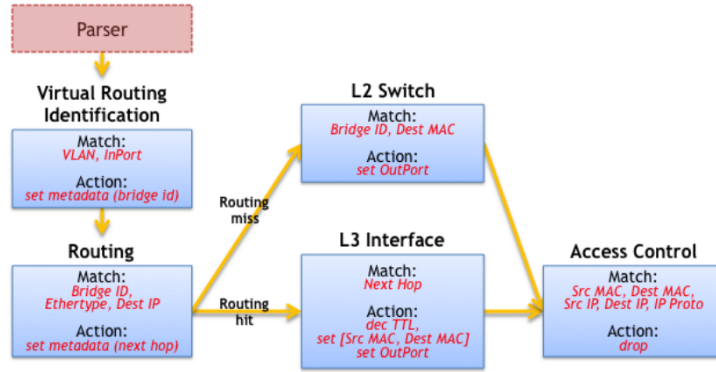


Figure 3.1. Table dependency graph for an L2/L3 switch. Reproduced from [2]

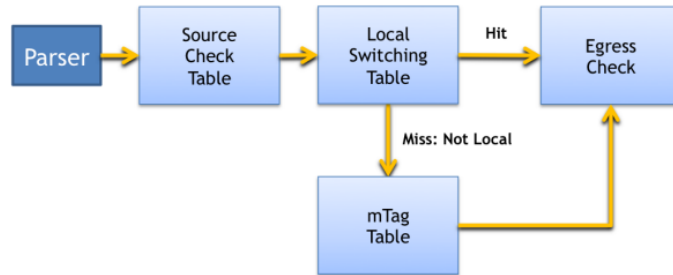


Figure 3.2. Flow chart for the mTag example. Reproduced from [2]

```
control main() {  
    // Verify mTag state and port are consistent  
    table(source_check);  
  
    // If no error from source_check, continue  
    if (!defined(metadata.ingress_error)) {  
        // Attempt to switch to end hosts  
        table(local_switching);  
  
        if (!defined(metadata.egress_spec)) {  
            // Not a known local host; try mtagging  
            table(mTag_table);  
        }  
  
        // Check for unknown egress state or  
        // bad retagging with mTag.  
        table(egress_check);  
    }  
}
```

Figure 3.3. Pseudocode for the mTag example Reproduced from [2]

escalating demands for real-time data and application delivery, coupled with the expansion of network infrastructures. It underscores the challenges associated with managing complex, fixed-function network equipment in the face of rapidly growing network sizes.

The authors emphasize the benefits of SDN in addressing these challenges and fostering network simplification and improved performance. They pose thought-provoking questions that the networking community is currently grappling with, laying the groundwork for the paper’s exploration.

The paper then delves into a detailed evaluation of SDN, starting with the OpenFlow protocol, which has played a pivotal role in configuring SDN-enabled devices. It discusses the physical separation of control and data planes in SDN, highlighting its advantages and applications in Network Function Virtualization (NFV) and network services development.

Crucially, the paper bridges the gap between SDN and P4, addressing the question of why SDN requires the P4 language. It underscores the limitations of the OpenFlow protocol and the need for P4’s upgradability, which enables the addition of functions and protocols to network devices through software updates, rather than hardware replacement.

The paper then pivots to the core concept of data plane programmability, defining it as the ability of network devices to expose low-level packet-processing logic through standardized APIs, enabling systematic reconfiguration. It touches upon various aspects of data plane programmability, including SDN with data plane programmability, data plane architectures, abstractions, network monitoring, P4 challenges, and applications.

Notably, the paper highlights the significance of data plane programmability in network monitoring, emphasizing its suitability for applications such as network metering, measurement, and debugging due to its direct and rapid access to network data.

The challenges associated with P4 language are discussed, with an emphasis on the importance of abstractions, consistency mechanisms, global state handling, and the distinction between dynamic behavior and static semantics. The paper underscores that P4 has proven effective in addressing these challenges.

Finally, the paper explores the diverse applications of data plane programmability, including telemetry, data processing, machine learning, load balancing, and resource pooling.

In conclusion, 'A Survey on P4 Challenges in Software Defined Networks: P4 Programming' provides an invaluable resource for comprehending P4, SDN, and data plane programmability. It serves as a comprehensive guide for understanding the development, current status, and research directions in this critical area of network technology.

In [4], the authors begin by highlighting the increasing challenges in network security due to the rapid evolution of technologies like IoT, AI, and cloud computing. They present statistics illustrating the rising threat landscape and emphasize the growing demands on network security infrastructure. The paper identifies limitations in traditional network security equipment, such as firewalls, which often lack the agility to adapt to emerging threats effectively. It then discusses how OpenFlow-based SDN solutions have been employed for network monitoring and attack detection but are hampered by centralized control and communication overhead issues. To address these challenges, the paper proposes the adoption of programmable data planes, specifically leveraging the power of P4. Programmable data planes offer advantages such as hardware-independent programmability and high-speed data processing capabilities, enabling the implementation of custom defense measures.

The authors delve into the capabilities of P4, including its ability to define data plane functions to support user-defined protocols and custom packet processing. They highlight the versatility of P4 in enabling advanced defense measures such as access control, heavy-hitter detection, and DDoS attack mitigation with minimal performance overhead.

While emphasizing the potential of data plane programmability, the paper also acknowledges its limitations. It serves as a foundational reference for understanding the evolution of network programmability, particularly in the context of SDN as you can see in Figure 3.4.

Furthermore, the paper explores P4 in-depth, covering its versions, workflow details as you can see in Figure 3.5, and the compilation process by the P4 compiler (3.5 - 2). It provides insights into how P4 programs define data plane behavior (3.5 - 1) and their deployment on various hardware platforms, including FPGAs.

The paper exemplifies the application of P4 in both academic and industrial settings, with a focus on traffic management, routing, forwarding functions, and, notably, network security. It underscores the advantages of P4-based programmable data planes in comparison to other approaches in the realm of network security which is the subject of this paper.

In [5], the authors explore the intricacies of service chaining and how precise control



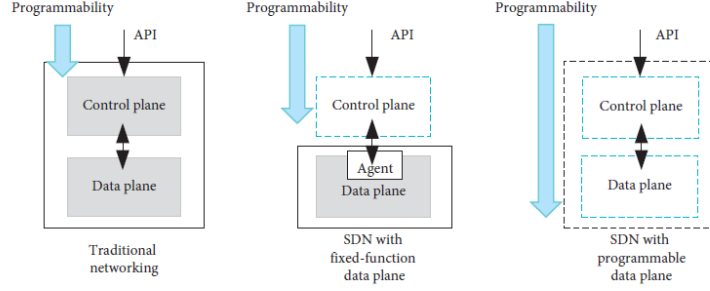


Figure 3.4. Traditional network, a fixed-function data plane SDN, and a programmable data plane SDN. Reproduced from [4]

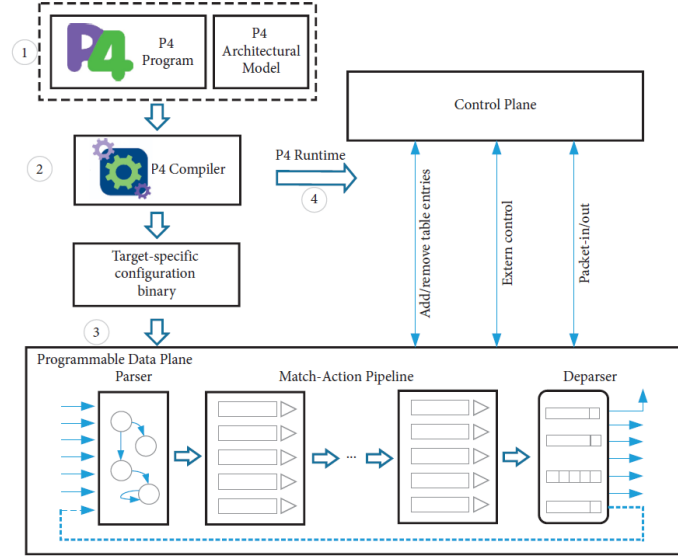


Figure 3.5. Workflow model based on P4 programmable data plane. Reproduced from [4]

over packet latency can be achieved by leveraging data plane programmability. The paper commences by establishing its primary motivation, which revolves around ensuring high-quality service delivery, particularly with regard to guaranteeing end-to-end latency for time-sensitive applications.

The paper emphasizes that latency within a service chain in a data center can be influenced by two main factors: intra-server virtualization inefficiencies, which result in delays related to the delivery of Virtual Network Functions (VNFs), and congestion events occurring at intermediate network elements that interconnect these VNFs.

To address these challenges, the paper advocates for the use of P4 and programmable switches. It argues that effective latency control within service chains necessitates access to stateful information, such as per-packet flow delay measurements, a level of granularity

that is often unavailable in traditional network switches.

The paper proposes two distinct solutions, both based on P4 pipelines, for controlling latency within service chains: one for inter-rack service chains and another for intra-rack service chains. The overarching strategy is to prioritize time-constrained flows and ensure that their delays remain below predetermined thresholds. Importantly, the research takes into account scenarios with and without congestion.

Following this introduction, the paper delves into network service function chaining (SFC) and its relation to Virtual Network Functions (VNFs). It also highlights the key features of VNFs, such as their roles in traffic engineering and security, while acknowledging the limitations of virtualized services in cloud environments.

The paper identifies a key challenge in the network domain, which is the presence of a single point of processing due to a central controller that handles all packet management responsibilities and policy definitions.

The introduction sets the stage for the exploration of data plane programmability concepts, which are fundamental to the paper’s proposed solutions. It underscores the use of programmable switches in this context and their role in achieving fine-grained per-packet treatment without relying heavily on central controllers.

The paper introduces two use cases: Use Case 1 (UC1) and Use Case 2 (UC2). UC1 considers a standalone scenario with a single P4 switch connecting the relevant VNF chains, while UC2 involves multiple decentralized switches cooperating to serve various VNFs and employing in-band network telemetry (INT) solutions. (as you can see in Figure 3.6 that is a combination of both scenarios)

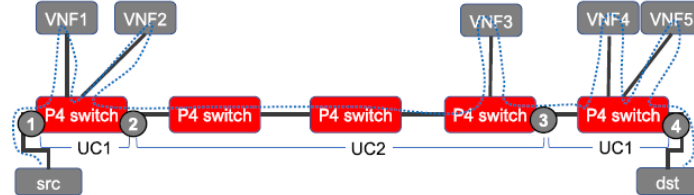


Figure 3.6. Combination of UC1 and UC2 for SFC latency segment control. Reproduced from [5]

The core idea in UC1 is that different VNFs are connected to the same switch (see Figure 3.7). To compute the latency that flows experience while traversing various VNFs, the switch uses an ingress timestamp when the packet arrives from one VNF (interface 1 in Figure 3.7) and an egress timestamp when the packet exits all VNFs and reenters the switch (interface 2 in Figure 3.7). The difference between these timestamps represents the latency experienced within the VNFs. Additionally, the switch measures and predicts the time packets spend waiting in queues, which can increase when congestion occurs, thus affecting latency. The final latency calculation combines these factors.

To manage latency, the paper introduces two thresholds: Priority Latency (PL) and Drop Latency (DL). Packets with latency below PL are treated normally, while packets

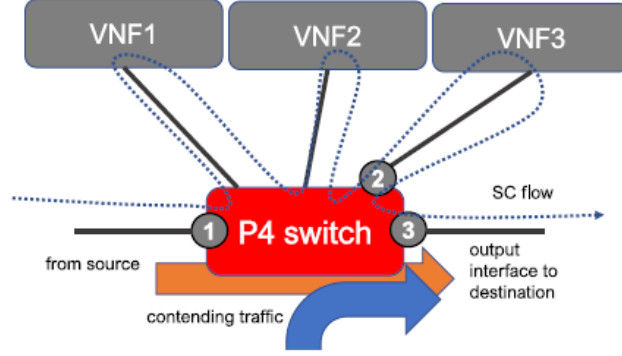


Figure 3.7. stand-alone switch scenario. Reproduced from [5]

with latency between PL and DL are given higher priority for faster delivery. Packets exceeding DL are considered out of date and are dropped.

The paper further details the algorithm and tables required in P4 switches to implement this latency control mechanism. (The complete implementation is observable in Figure 3.8)

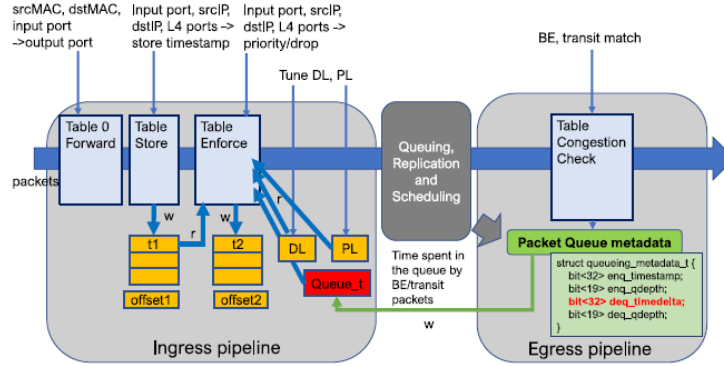


Figure 3.8. Use Case 1: P4 pipelines implementation. Reproduced from [5]

In UC2, the approach is extended to encompass multiple switches along the route. The paper introduces the concept of an INT-enabled domain, where switches can insert extra headers into selected traffic for switch metadata purposes. The first node adds an INT extra header to the packet, and the last node extracts this header and uses its information. The methodology in UC2 essentially builds upon that of UC1, with the addition of INT headers to capture relevant data. You can see a general view of this scenario in Figure 3.9.

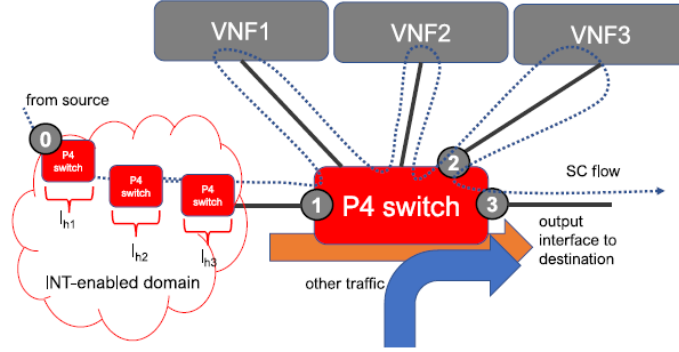


Figure 3.9. Use Case 2: distributed INT switch scenario. Reproduced from [5]

This study considerably contributes to our research by offering a valuable understanding of the use of headers and metadata within programmable switches for extracting more information from packets. Although our thesis does not expressly address service chaining or Network Function Virtualization (NFV), our major focus is on P4 switches. The broad use of P4 switch features and methodology in this research provides excellent direction, assisting us in furthering our aims in monitoring latency and other characteristics inside programmable network devices.

Furthermore, this article serves as a useful resource in our attempt to design functionality for programmable switches' ingress and egress pipelines. The paper's step-by-step approach to the exploitation of multiple pipelines and how the programmable switch manages queues is extremely fascinating. This advice is critical to the advancement of our research activities, allowing us to work toward the precise goals mentioned in our thesis.

The paper [6], presents a pioneering approach to addressing the critical issue of establishing consistent order among operations in distributed systems. Traditionally, achieving such ordering relied on application-level protocols like Paxos or two-phase locking, but these methods came with inherent limitations, including scalability challenges, fault tolerance issues, and load balancing problems due to the involvement of a single sequencer.

The Hydra protocol, introduced in this paper, revolutionizes network ordering by leveraging a distributed group of sequencers. This innovative approach employs weakly synchronized clocks, unique identification numbers for each sequencer to detect message drops, and periodic timestamp messages to ensure progress, even when some sequencers are idle. The results showcase Hydra's superior performance compared to traditional approaches, boasting significantly higher scalability, shorter sequencer failover times, and improved network-level load balancing.

The paper also highlights the pressing need for data replication in data centers and the historical challenges posed by consensus algorithms like Paxos, Viewstamped Replication, and Raft, which introduced performance and latency overhead. Hydra comes to the rescue by routing requests through sequencers, enabling the implementation of lighter-weight

consensus protocols through in-network processing. This groundbreaking innovation effectively mitigates the associated costs. Unlike conventional network sequencing, which often involves serialization with scalability challenges, routing complexities, and compatibility issues with application-level recovery mechanisms, Hydra introduces an ingenious approach. It allows packets to be sequenced through multiple active sequencers implemented in programmable switches, streamlining coordination and efficient message drop detection.

The deployment strategy employed by Hydra includes both a software implementation running on end hosts and a P4-based implementation running on an Intel Tofino programmable switch. This P4-based approach is of particular interest to our research, as it demonstrates how programmable switches can be used to develop the functions necessary to achieve the paper’s objectives.

The paper provides statistical evidence demonstrating Hydra’s superiority in packet ordering performance compared to other protocols. It then delves into the traditional approach to packet ordering in networks, emphasizing the drawbacks of relying on a centralized sequencer. The limitations of this method include scalability bottlenecks, prolonged system downtime, deteriorated data center network properties, and incompatibility with multi-pipeline switches.

Hydra introduces the concept of sequencing with multiple sequencers and outlines deployment options that span end hosts, top-of-rack switches, root switches, and sequencer appliances. Regardless of the deployment option, Hydra seamlessly integrates with existing data center routing structures. Each deployment variant is allocated a distinct IP address and dynamically adaptable routes to account for sequencer changes.

The core abstraction provided by Hydra is a group communication protocol, offering properties such as partial ordering, unreliable delivery, and drop detection. The paper proceeds to describe the Hydra algorithm in detail, including the use of physical clocks for message ordering, combining physical clocks and multi-stamps for drop detection, ensuring progress with flush messages, and handling sequencer failures. I do not enter the details of these processes. I just put Figure 3.10 from the paper that explains the complete algorithm of Hydra with an example. You can see a short explanation of this algorithm in the capture of the figure.

The most intriguing aspect for our research lies in the efficient implementation of Hydra sequencers using programmable switches, facilitating in-network sequencing. Hydra deploys a dynamic set of groupcast senders, receivers, and sequencers managed by a configuration service. Centralized SDN or source routing approaches are used to manage groupcast routing, allowing for efficient routing to randomly selected reachable sequencers. Hydra sequencers maintain a minimal state, including a unique sequencer ID, a sequence number for each receiver group, and a monotonically increasing physical clock.

Hydra groupcast is implemented as an application-level protocol atop UDP, using a customized Hydra header to facilitate sequencing. Programmable switches play a crucial role in this implementation by utilizing switch register arrays to store sequence numbers for receiver groups and efficiently process packet sequencing in parallel. This approach enables Hydra to scale to a higher number of groups while maintaining efficient processing.

In conclusion, this paper is a valuable resource for our research. It offers insights into



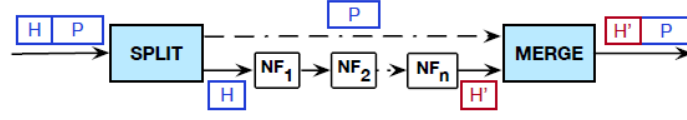


Figure 3.11. Abstract PayloadPark deployment. Split decouples the packet into header H and payload P. Shallow NF chain NF1, NF2 ... NF<sub>n</sub> transforms header H into H'. Merge reassembles the header H' with payload P. Reproduced from [7]

data plane. "Merge" reunites the potentially modified header from the network function chain with the payload before forwarding the packet to its destination. These operations are activated on a per-port basis, with "Split" playing a central role. In this operation, a unique tag is assigned to the payload, allowing it to be detached from the packet header and securely stored within the data plane. A distinctive PayloadPark header is appended, including an "Enable" bit indicating successful payload storage. The "Merge" operation complements this process by reuniting the payload with the modified header obtained from the network function chain. The assigned tag serves as a crucial reference point for locating the stored payload, which is then seamlessly merged with the header. The PayloadPark header is subsequently removed, and the associated payload-consumed space is efficiently reclaimed. The PayloadPark headers that are added to the packet header are observable in Figure 3.12.

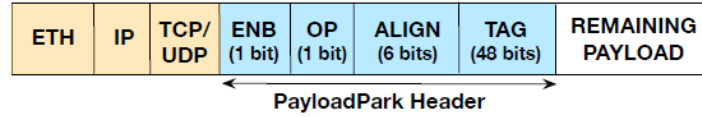


Figure 3.12. PayloadPark header. The Enable (ENB) bit indicates whether the PayloadPark operation is enabled. Opcode (OP) indicates the operation to be performed: Merge | Explicit Drop. ALIGN bits are for byte alignment. TAG is a unique identifier for the packet. Reproduced from [7]

The paper highlights that PayloadPark has become feasible thanks to newly available Reconfigurable Match-Action Table (RMT) switches. These switches are equipped with programmable ASICs that enable the implementation of in-network optimizations using domain-specific languages like P4. However, RMT switches come with limitations in terms of storage resources and per-packet compute and stateful operations, aimed at ensuring packet processing at line rate.

The research then delves into the details of the PayloadPark algorithm. The architecture of PayloadPark includes critical components such as a packet tagger, a lookup table, and a payload evictor. These components ensure the robust functioning of PayloadPark, with the payload evictor responsible for recovering memory occupied by lost or dropped payloads following the "Split" operation. PayloadPark has been thoughtfully designed to

operate within the constraints of limited storage resources while ensuring efficient network function. You can see the packet flow in PayloadPark in Figure 3.13.

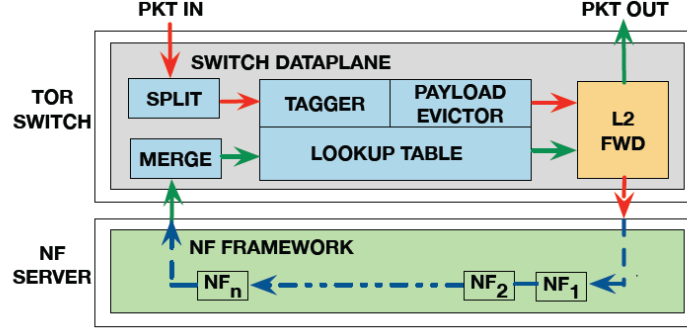


Figure 3.13. Packet flow in PayloadPark. Split decouples the packet header and payload, and stores the payloads in the lookup table. The Merge operation merges the headers from the NF server with the payloads stored in the lookup table. L2 FWD forwards packets using L2 forwarding. Reproduced from [7]

One key aspect of our research lies in how they have implemented this algorithm using the programmable switch data plane (see Figure 3.14). PayloadPark’s versatility is demonstrated through its integration into the switch data plane architecture, with a focus on the "Split" operation. This operation generates a tag consisting of two components: an index (TI) used to locate a vacant storage location for the packet payload and a generation number (CLK) that distinguishes between evicted and non-evicted payloads. The metadata table plays a crucial role in determining index availability. If the index is empty, the CLK value and a predefined expiry threshold (EXP) are written into the table. The PayloadPark header is then affixed to the packet, with the ENB and tag values duly updated. If the lookup table entry is occupied, the ENB bit is set to zero, and a 2-byte CRC is included. Payloads are stored in a dedicated payload table, structured as a two-dimensional array across various memory address tables (MATs), with the exact payload storage capacity varying across switches.

The evaluation section of this research investigates the efficiency of the PayloadPark algorithm. PayloadPark’s approach is notable for its efficient management of memory resources. The "Merge" operation distinguishes between evicted and non-evicted payloads using the packet tag and effectively reclaims memory by cleaning up long-lived payloads. The system efficiently handles memory reclamation, ensuring optimized network performance.

The validation process forms a critical stage within the PayloadPark workflow, where stored payloads are rigorously assessed. This process enhances the reliability and accuracy of payload integration into validated packets, ensuring the overall robustness of the PayloadPark system.

In summary, this paper provides valuable insights into addressing the inefficiencies in network function deployment using P4 and programmable switches. It offers a comprehensive overview of the PayloadPark mechanism, including its architecture, operations,



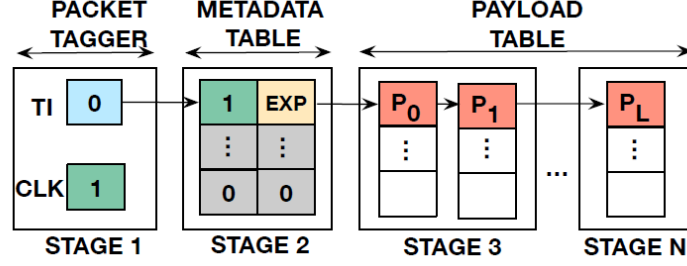


Figure 3.14. PayloadPark data plane implementation. The tagger has registers for the table index (TI), which is an index into the lookup table, and a clock (CLK). The metadata table contains two values at each index, the value of the clock when the index was occupied and an expiry threshold (EXP). If the index is available for storing payload, its EXP value is 0. Payload Blocks ( $P_0, P_1 \dots P_L$ ) are striped across MATs. Reproduced from [7]

and implementation in programmable switch dataplanes. This information aligns with our research objectives related to data plane programmability.

The paper [8], focuses on addressing the challenges associated with high-throughput packet processing in programmable network switches, particularly within the context of the RMT programmable high-throughput switch architecture. The authors introduce a method called PRECISION, which leverages Probabilistic Recirculation to identify top flows on a switch. This technique optimizes access to stateful memory while complying with RMT constraints, outperforming previous heavy-hitter detection methods that avoided recirculation. PRECISION offers flexibility and high throughput, making it valuable for applications such as load balancing and traffic engineering.

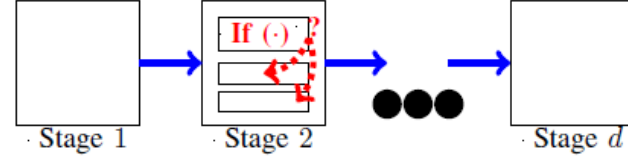
The paper underscores the significance of programmable network switches in deploying network algorithms like traffic engineering, load balancing, quality-of-service optimization, anomaly detection, and intrusion detection. It emphasizes that accurate measurement capabilities are crucial for these applications as they extract valuable information from traffic for informed decision-making.

The core challenge addressed in this paper is the limitation of SRAM memory for measuring at a 100 Gbps line rate. Traditional heavy-hitter algorithms attempt to overcome this limitation by storing flow state information for the largest flows. However, this approach introduces a trade-off between memory space and accuracy. The paper mentions two categories of solutions for heavy-hitter detection: counter-based and sketch-based algorithms.

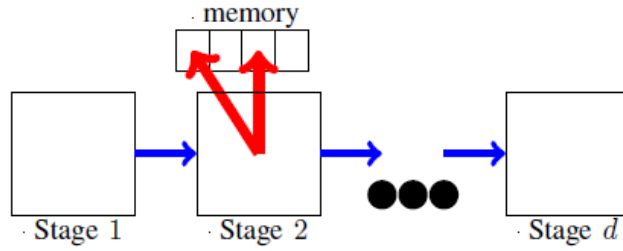
Counter-based algorithms maintain a bounded-size flow cache and can directly solve the top-k problem. In contrast, sketch-based algorithms implicitly share counters among multiple flows and require additional efforts to address the top-k problem. While sketch algorithms are implementable in programmable switches, the paper highlights that supporting top-k measurements motivates deploying counter-algorithms, which present significant challenges due to the restrictions imposed by high-performance packet processing.

The Reconfigurable Match Tables (RMT) switch programming model is crucial in

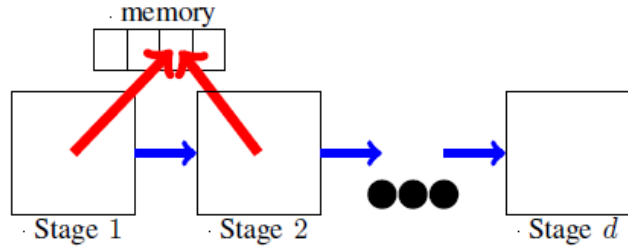
designing measurement algorithms. However, this model introduces limitations such as limited branching, concurrent memory access, single-stage memory access, and a fixed number of stages, which the paper details. You can see an illustration of some of these restrictions in Figure 3.15 that is provided by the paper. It then introduces the PRECISION algorithm, emphasizing its compatibility with the RMT high-performance programmable switch architecture.



(a) Restriction I: Limited in-stage branching



(b) Restriction II: Limited memory access within a stage



(c) Restriction III: One memory address cannot be accessed from multiple stages

Figure 3.15. Illustration of some restrictions imposed by RMT pipeline model for designing measurement algorithm. Reproduced from [8]

PRECISION employs probabilistic recirculation, randomly recirculating a small portion of packets from unmonitored flows. When a packet is recirculated, it traverses the programmable switching pipeline twice. In the first pass, the paper attempts to match a packet to an existing flow entry; if successful, the counter is incremented. If there's no match, the packet is probabilistically recirculated to claim an entry with the new packet's flow ID. This approach simplifies memory access without significantly degrading throughput while maintaining high monitoring accuracy.

The paper then compares PRECISION to previous algorithms in the field of heavy-hitter detection, emphasizing its advantages and innovations.

Subsequently, the paper delves into a precise definition of the two heavy-hitter problems investigated: the frequency estimation problem and the top-k problem. It also defines the concept of a flow in a network based on packet headers and discusses existing approaches to solving these problems.

The main portion of the paper focuses on the design and implementation of the PRECISION algorithm. It addresses how PRECISION deals with the restrictions imposed by the RMT switch architecture. One crucial point for your research is how PRECISION simplifies memory access within RMT switches. The paper highlights that implementing PRECISION is particularly challenging due to the need to replace an entry after knowing the minimum sampled counter value, which is only available after reaching the end of the pipeline. To overcome this challenge, the paper utilizes the recirculation feature on switches, allowing packets to traverse the pipeline again and removing conditional branching for register access. This approach facilitates more versatile packet processing at the expense of packet forwarding performance.

The paper also explains how recirculated packets are managed and how metadata is used to distinguish between recirculated packets and regular packets. This recirculation function implemented in programmable switches is a focal point of interest for my research, as it differs from our approach, which involves recirculating all packets or customized packets. The approach taken in this paper regarding recirculation in programmable switches can serve as a foundational reference for my research on developing our own version of recirculation in programmable switches.

The paper [9] proposes congestion avoidance approaches that use entirely programmable data planes to reduce end-to-end latency. The programmable switches can monitor latency-critical flow processing and queuing delays and respond quickly to congestion by rerouting affected flows. They employ programming languages such as P4. The data-plane method reduces jitter, average, and maximum latency as compared to non-programmable approaches. For reliable Internet traffic, consistent feedback and a maximum delay restriction are required. To decrease end-to-end latency, packets should not be delayed or rejected by the network. Network delays include propagation, transmission, processing, and queuing. A trustworthy system necessitates that these delays be maintained under control and to a minimum.

The research digs into the challenging obstacle of network congestion among nodes, a widespread problem that causes queuing delays. Traditional congestion control technologies, such as TCP, need round-trip time for congestion detection, which is often conducted at sender nodes. Introducing a new paradigm, software-defined networking (SDN) enables more flexible adaptations to constantly changing network circumstances via centralized controllers. While SDN has the potential to improve Quality of Service (QoS) routing and traffic prioritization, it requires continual monitoring and recalibration. The key problem is to enable forwarding nodes to proactively manage and anticipate congestion rather than depending on source-based or controller-driven mitigation. A key component of the study's strategy is identifying and responding to rising delays at the switches themselves, which represents a strategic change from source-based intervention.

This method is critical for achieving excellent end-to-end latency and jitter reduction. Programmable switches, as well as domain-specific programming languages like P4, have proven to be important tools for this purpose. A hierarchical system is described, with a local congestion control module monitoring low-latency flows and a central controller modulating latency thresholds to provide an effective congestion detection and avoidance mechanism.

The study provides a separate "Congestion Detection and Avoidance" module, as you see in Figure 3.16, suited for latency-critical flows inside this architecture. This module monitors processing and queuing delays and automatically redirects traffic to alternate paths or sends alarms to preceding nodes when delay components increase. A key finding from the study is that, in accordance with P4 language rules, rerouting requires controller interaction within the data plane. Register-based techniques are used, coupled with intelligently maintained meta-data, to ensure interoperability with all P4-compliant devices.

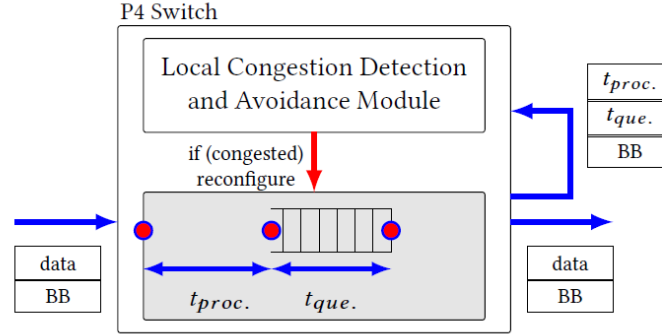


Figure 3.16. Detection of congestion in the data plane. Every switch has a small congestion avoidance module, gathering statistics (processing and queuing delay). Reproduced from [9]

The research addresses the difficulties that arise from the concurrent processing of packets from the same flow, a circumstance that can worsen processing delays and lead to race conditions. To address these issues, the method employs techniques such as packet copies and packet digests. While both techniques have benefits, the research takes a different approach by transferring congestion monitoring inside a P4 program and using digests to notify the local module when delays exceed specified levels. This technique reduces the possibility of overloading the local control application module and provides the freedom to adjust parameters such as the number of subsequent packets with rising delays, as well as queuing and processing delay thresholds.

The full findings of the study on "Fast Network Congestion Detection and Avoidance Using P4" are linked to the main subject of our thesis, "Delay Control with Programmable Data Planes." The study's original methodology and innovative approaches to congestion mitigation and network performance are critical to our research. The use of programmable

data planes, notably P4, and its accompanying approaches, has the potential to enhance our research into latency control. The study’s emphasis on proactive detection, targeted intervention, and dynamic rerouting solutions aligns with our efforts to design more efficient and responsive networks.

The study [10] delves into a prevalent issue known as Bufferbloat, a condition characterized by the presence of oversized packet buffers within forwarding equipment, which consequently engenders excessive latencies within networks. The research reveals that this challenge is effectively mitigated through the application of Active Queue Management (AQM) algorithms, such as CoDel and PIE. These algorithms, which exhibit the capacity to dynamically adjust buffer sizes and alleviate buffer bloat, have demonstrated their effectiveness in enhancing network performance. Notably, the study underscores a pertinent limitation wherein contemporary network equipment often lacks access to these AQM algorithms, potentially hindering their broader deployment.

A pivotal contribution of the study revolves around the implementation of an open-source CoDel solution within the P4 framework. This implementation has far-reaching implications, especially in contexts such as traffic shaping within ISP access networks. The study expounds that Bufferbloat arises due to the accumulation of excessive-sized packets within a queue as active connections strive to reach TCP window sizes or fill the queue to its brim, leading to packet drops and subsequent latency spikes. Such latency spikes disproportionately affect latency-sensitive traffic like voice-over IP, necessitating effective mechanisms to mitigate them.

The CoDel and PIE AQM algorithms, highlighted by the study, introduce dynamic feedback mechanisms that intelligently manage queuing delays. Particularly, the CoDel algorithm, as elaborated in the research, leverages TARGET and INTERVAL parameters to meticulously control the periodic queueing delay, ensuring it remains consistently beneath the predefined TARGET threshold. This approach, as elucidated by the study, prevents packet losses and effectively extends the intervals until the desired TARGET delay is achieved.

A significant achievement of the study is the seamless integration of the CoDel algorithm into the P4 reference pipeline. This integration encompasses both packet processing and queue storage, thereby enabling efficient management of queuing delays. Importantly, the research underscores the accessibility and applicability of this solution by revealing its compatibility with the P4 reference model BMv2. Moreover, the implementation’s versatility is highlighted, as it can operate on Linux-based computers without necessitating specialized hardware or costly proprietary software.

The study further emphasizes the incorporation of packet timestamping, a crucial aspect facilitated by the BMv2 architecture. This feature provides essential access to queueing delay information, thereby contributing to more effective queue management strategies. In essence, the study illuminates the potential of P4-CoDel as a transformative solution for alleviating Bufferbloat and enhancing network performance, particularly in scenarios demanding latency-sensitive traffic management.



## Chapter 4

# Delay Control by Packet Recirculation

The primary focus of my thesis centered around investigating the feasibility of employing packet recirculation within programmable switches to control delay. The underlying motivation was rooted in a fundamental question: Could we keep an initial packet within the switch until a subsequent packet from the sender arrives, and then intelligently forward the first packet after the later packet? In pursuit of this goal, I embarked on a two-phase journey. The first phase involved the development of a packet recirculation mechanism within the BMv2 switch. Initially, I implemented a general recirculation process, wherein the switch recurrently processed all incoming packets for a predefined number of cycles. To facilitate this, I introduced a packet metadata element, which is an incremental counter that increases each time a packet re-enters the switch. This counter guides the recirculation process until the specified cycle count is achieved. Further refinement led to a more sophisticated approach, where only specific packets were recirculated. I extended the switch’s capabilities by integrating additional instructions to identify and selectively process packets with custom headers. This development allowed the switch to efficiently differentiate between regular and custom packets, applying the recirculation algorithm only to the custom packets while forwarding the normal ones promptly or discarding them as needed.

The second phase involved comprehensive testing to evaluate the effects of the recirculation algorithm on packet delay and network throughput. To ensure robustness and relevance, experiments were conducted in both virtual and real environments, employing the Linux-based operating system. In Linux, using Mininet which is a powerful network emulation tool, I created experimental topologies comprising source and destination hosts interconnected by BMv2 switches.

In these experiments, I examined diverse scenarios. Initially, I focused on a single switch topology to evaluate the influence of packet recirculation on delay and throughput in a straightforward setting. Subsequently, I extended the investigation to include multi-switch topologies, exploring the impact of recirculation within more complex network configurations.

To quantify delay, I made use of the well-known ‘ping’ utility, which measures the

Round Trip Time (RTT) between source and destination by sending ICMP echo request and reply packets. This enabled me to analyze how recirculation affected packet delay under various conditions. For evaluating throughput, I turned to the renowned 'Iperf' tool, capable of measuring the maximum achievable network throughput. I conducted experiments utilizing both TCP and UDP protocols, adjusting parameters like recirculation cycles, link bitrate, and sending rates to comprehensively assess their effects on network performance.

In summary, my work encompassed the design and implementation of a novel packet recirculation mechanism within the programmable switch, followed by an extensive testing phase to explore its impact on delay and throughput in diverse network scenarios.

## 4.1 Packet Recirculation development

In this section, I will explain how the recirculation algorithm works in programmable switches and detail the development of the recirculation algorithm in these switches. It is critical to understand how the control plane and data plane of the programmable switch should function together to enable recirculation. I'll go over each of them. It is essential to remember that, as previously said, we should program the data plane using the p4 language. The first strategy that comes to mind and is easier to implement is to recirculate all received packets a specific number of times. The next step will make the recirculation algorithm more specific, allowing more control over the mechanism and allowing the programmable switch to recirculate custom packets as desired.

### 4.1.1 Strategy 1

Let's start with a basic yet effective strategy: the switch recirculates all incoming packets for a certain amount of cycles. The process begins when a packet arrives and encounters the parser, which is a skilled entity that extracts fields from the packet's header without assuming their meanings. These fields form the foundation upon which the switch's subsequent actions and matches are predicated. In the switch's architecture, a state machine navigates the headers' fields from start to finish. P4 describes this state machine as a constellation of transitions, each triggered by fields in the current header. Upon reaching a state for a new header, the state machine extracts the header using its specification and proceeds to identify its next transition. This parsing commences in the start state, where the parser unravels the Ethernet header. Depending on the Ethernet type, it moves forward, potentially unlocking the IPv4 state, where the IPv4 field emerges into view. This is all of our transition in this state machine. For advancing the recirculation strategy, we do not need any extra work in parsing the packets. The switch parses the packets in such a way that it is just a simple L2/L3 forwarding device for forwarding packets. Now, when these extracted fields of the header are in hand, they traverse the ingress pipeline. In our architecture, the ingress pipeline is responsible for doing anything that is needed for forwarding the packets to the destination, so the match tables and actions are defined here based on this responsibility. Which action can be performed on a pipeline that is responsible for forwarding? it is forwarding, or if it cannot forward the packet, drop it. So we define these two actions in this pipeline. The match should be done based



on the IPv4 header that we extracted from the packet. The forwarding is done based on the destination address part of the IPv4 header, and the switch makes this forwarding available if there is a match in this field, which is the IP address of the destination. If there is no match between this field of the packet header and the forwarding rules that the controller provides for the switch, it will drop the packet. The forwarding action's first act involves determining the egress port which is the gateway to the destination. Like travelers adapting to different landscapes, packets' MAC addresses adjust, and their TTL counts diminish as they traverse each hop. The next scene unfolds within the egress pipeline that is responsible for recirculation action. The strategy is a recirculation for every packet, but only for a predefined number of cycles, so no matching table is defined in this pipeline. Here, metadata enters the picture, which is an element bestowed upon packets to convey their history. For the first encounter with the egress pipeline, this metadata is zero and is waiting to be increased. Just consider that the number of recirculations for each packet is determined by a constant value that I defined previously. As the packet arrives, the egress pipeline investigates this metadata. If it detects a count of zero, it invokes a predefined recirculation function that is programmed into the switch. This function marks the beginning of a packet's cyclic journey, simultaneously advancing the metadata count. With each cycle, this counter ascends until it meets the initial recirculation constant. Just for better clarification, the egress pipeline just acts when the counter metadata of the packet is less than the number of recirculation constant value; after that, it does nothing, and the packet is forwarded as indicated in the ingress pipeline.

#### **P4 program for defining the packet-processing pipeline in the first strategy:**

The provided P4 program serves as a model for defining the packet-processing pipeline within a programmable switch when the switch recirculates all the packets for a definitive number of cycles. This program outlines how incoming network packets are processed, recirculated, and forwarded based on their headers. Let's break down and elaborate on the key components of this program:

- **Headers and metadata:**

The program begins by defining various headers that include critical information about the incoming packets, such as Ethernet and IPv4 headers. These headers mirror the structure of the respective protocol specifications. Additionally, a metadata structure is introduced to store essential metadata, including a recirculation counter (Figure 4.1).

- **Parser:**

You can see the code of the parser in Figure 4.2. As I explained before, the parser is an entity that extracts fields from the packet's header.

- **Deparser:**

```

/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
#define RECIRCULATE_TIMES 100

/***** H E A D E R S *****/

typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

/* TCP/IP Headers */

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> tos;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    @field_list(1)
    bit<8> counter;
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
}

```

Figure 4.1. Defining headers in the p4 file

The Deparser ensures that the modified headers are reassembled into the outgoing packet format before transmission. (see Figure 4.3)

- **Checksum verification:**

The checksum verification control block validates the integrity of packet headers.

- **Ingress processing:**

As discussed before, the ingress pipeline is responsible for packet-forwarding decisions. It defines actions like packet dropping and forwarding. The ingress control block includes a table called `ipv4_lpm` that matches the destination IP address of the IPv4 header. Depending on the match result, packets can be forwarded, dropped, or subjected to a "no action" outcome. (see Figure 4.4)

**Forwarding process:** The process of forwarding within a programmable switch is a critical mechanism that determines how incoming packets are directed to their intended

```

/***** P A R S E R *****/
parser MyParser(packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {
    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            TYPE_IPV4: ipv4;
            default: accept;
        }
    }
    state ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}

```

Figure 4.2. The development of Parser in the p4 file. The parser is responsible for extracting information from the incoming packet headers. It employs a series of states to identify the type of packet (Ethernet, IPv4, etc.).

```

/***** D E P A R S E R *****/
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

```

Figure 4.3. The development of Deparser in the p4 file. The deparser reassembles the headers into the outgoing packet.

destinations. This process is overseen by the MyIngress control block, operating within the switch’s ingress pipeline. It ensures efficient data flow across the network.

At the core of this process are the Ethernet header and MAC addresses. The Ethernet header contains two vital pieces of information: the source MAC address, identifying the sender, and the destination MAC address, specifying the receiver. These MAC addresses are fundamental for proper communication, guiding packets to their correct endpoints. In the MyIngress control block, the `ipv4_forward` action plays a key role by adjusting these MAC addresses to dictate the packet’s path through the switch.

However, the transmission of packets is done like what happens in the IP protocol. The IPv4 header holds the source and destination IP addresses, needed for routing packets between networks. The `ipv4_forward` uses the destination IP address as a reference point. This reference determines the action to take, whether forwarding the packet, discarding it, or taking no action, based on interactions with the `ipv4_lpm` table.

The Time To Live (TTL) parameter, embedded in the IPv4 header, adds an essential

```

/***** INGRESS PROCESSING *****/
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}

```

Figure 4.4. Instructions for programming the Ingress pipeline in the programmable switch; responsible for forwarding the packet to the destination using Ethernet and IP headers

dimension. The TTL sets a limit on a packet's lifespan. It starts when the sender initiates the packet and decreases with each switch it encounters. The TTL prevents packets from staying indefinitely or circulating in loops. The `ipv4_forward` action lowers the TTL value, indicating the packet's progress through the network.

As the forwarding process progresses, the TTL ensures the traversal of the network. Its decrement signifies meaningful progress toward the destination.

In summary, the forwarding process, guided by Ethernet and IP headers, ensures seamless data flow, enabling effective communication within the network.

- **Egress processing:**

The egress pipeline focuses on packet recirculation within the switch. The `MyEgress` control block handles recirculation decisions. If the packet's recirculation counter is below a predetermined threshold (`RECIRCULATE_TIMES`), the packet is recirculated within the switch. (see Figure 4.5)

- **Checksum computation:**

The checksum computation control block calculates and updates checksum fields in the packet headers to ensure their validity. (see Figure 4.6 )

```

/*****
*****  E G R E S S   P R O C E S S I N G   *****
*****/

control MyEgress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    table recirculate {
        key = {
            meta.counter: exact;
        }
        actions = {
            NoAction;
        }
        size=1;
        default_action=NoAction();
    }

    apply {
        recirculate.apply();
        if (meta.counter < RECIRCULATE_TIMES){
            meta.counter = meta.counter + 1;
            recirculate_preserving_field_list(1);
        }
    }
}

```

Figure 4.5. Instructions for programming the Egress pipeline and the development of recirculation in the programmable switch; responsible for packet recirculation using packet metadata

- **Switch architecture:**

The program concludes by defining the architecture of the switch using the V1Switch construct. This encapsulates the parser, checksum, ingress, egress, and deparser components in the correct order to form the complete packet-processing pipeline. (see Figure 4.7)

### Switch Controller:

Now what should the controller do in this strategy? As I said before, the recirculation is done for all the packets, so the controller should not perform anything for the recirculation part. It should just determine how the forwarding should be done. A rule governs this path: match the IPv4 destination address. With this criterion in hand, the controller directs the switch with the parameters of forwarding instructions, including the port that the packet should exit for reaching the next node and the MAC address of the destination for each packet's progress.

Think of the control plane as the brain of the switch. It's a separate part that controls how the switch operates. In our setup, we have a control plane file that tells the switch what to do and how to act (see Figure 4.8), which we reference in our network topology to specify the properties and instructions that guide the switch on how to manage data

```

/***** C H E C K S U M   C O M P U T A T I O N *****/
/*****/

control MyComputeChecksum(inout headers hdr, inout metadata meta) {
    apply {
        update_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.tos,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}

```

Figure 4.6. Checksum computation instructions in programmable switch

```

/***** S W I T C H *****/
/*****/

//switch architecture
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;

```

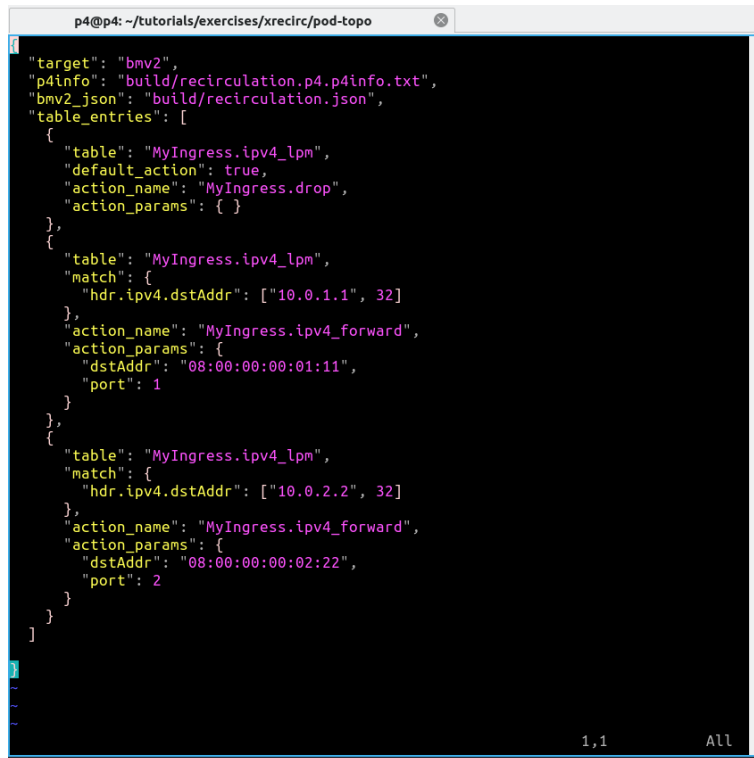
Figure 4.7. Switch architecture in programmable switch

traffic.

The control plane's instructions are designed to be clear and explicit, ensuring that packets are processed and routed accurately within the switch. By providing detailed rules and actions, we enable the switch to make intelligent decisions about how to handle incoming data.

The actions defined in the control plane are closely tied to the instructions in the data plane, which determine how the switch's hardware processes packets. Together, these two planes collaborate to ensure efficient and effective data forwarding based on the specified

conditions.



```

"target": "bmv2",
"p4info": "build/recirculation.p4.p4info.txt",
"bmv2_json": "build/recirculation.json",
"table_entries": [
  {
    "table": "MyIngress.ipv4_lpm",
    "default_action": true,
    "action_name": "MyIngress.drop",
    "action_params": { }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:01:11",
      "port": 1
    }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:02:22",
      "port": 2
    }
  }
]

```

Figure 4.8. The Controller of the switch in the state that the switch recirculates all the packets, the controller just indicates the match-actions for forwarding the packets

- **Switch type and log files:**

In the control plane file, we begin by specifying that the switch is of the BMv2 type. This distinction is important as it informs the network infrastructure about the type of switch being employed. Additionally, we provide addresses to log files that capture important switch-related information for monitoring and analysis.

- **Default behavior:**

The control plane includes instructions for the default action that the switch should take. If a packet's destination doesn't match any explicitly defined rules, the switch employs a default action. This can involve actions like dropping the packet to ensure network integrity.

- **Specific Destination Matching:**

In our specific configuration, we've defined rules to handle packets with certain destination IP addresses. For instance, if a packet's destination IP matches "10.0.1.1", the

control plane specifies an action named "ipv4\_forward." This action involves forwarding the packet to a designated port (Port 1) and setting the destination MAC address to "08:00:00:00:01:11". Similarly, if a packet's destination IP matches "10.0.2.2", the control plane triggers the same "ipv4\_forward" action, forwarding the packet to Port 2 with a destination MAC address of "08:00:00:00:02:22".

#### 4.1.2 Strategy 2

This process extends beyond the initial strategy. Within the complex framework, custom packets are introduced, forming a distinct segment. These packets are characterized by custom headers, primarily identified by their `content_id`. The parser's capabilities expand to accommodate these custom headers, orchestrating their progression. Subsequently, as the egress pipeline checks the metadata of these packets, the controller issues specific instructions. This involves `content_id`-based matching and the execution of actions tailored to the unique attributes of each custom packet. I begin the second method by building on the previous strategy's base. In this method, the parser's operation evolves to accommodate the new dynamics. It initiates the transaction in a start state and extracts the Ethernet field, similar to the previous strategy. However, a new aspect appears at this point. Depending on the Ethernet type within the header, the parser directs the transition to different states. If the Ethernet type corresponds to IPv4, it advances to a state-linked with IPv4. Conversely, if the Ethernet type corresponds to custom data, it progresses to a state designed for this specific type. This addition of a custom state augments the parser's ability to handle distinct data scenarios. Here, a custom header comes into play, appended to the packet, which in turn is extracted for matching, action execution, and processing. Upon completion of header parsing, the mechanism transitions to the processing stage. This phase, like the prior strategy, assigns packet forwarding duties to the ingress pipeline while assigning recirculation responsibility to the egress pipeline. The forwarding mechanisms within the ingress pipeline remain consistent with the previous strategy, with no additional further discussion. However, it is in the egress pipeline that the distinctions of this second strategy become evident. Here, the process involves verifying whether the packet contains custom data or not. In the presence of valid and parsed custom data, the packet assumes the classification of a custom packet. The next step is controlled by the predetermined number of recirculations, a constant parameter defined like the previous strategy. To track the number of recirculations, metadata is added to the custom data, denoted by `ingress_number` and `egress_number` fields within the custom data header. Further differentiation is achieved through a `content_id` field, addressing different types of custom data. The following method examines the availability of the custom data header in the egress pipeline. If present, the `ingress_number` field is incremented, signifying the packet's progress through the pipeline. Subsequently, a decision point emerges: to recirculate or forward the packet. If the number of times the packet has reached the egress pipeline is less than the defined recirculation constant, a recirculation function is executed. However, if forwarding is indicated, the packet's `content_id` is employed to determine its destination. This distinction is reflected in different IP addresses, one for each `content_id`, along with corresponding port numbers for forwarding. The reasoning for this distinction is to treat custom packets differently from



normal packets. Custom packet forwarding is defined based on the `content_id`, setting the foundation for a customized and relevant forwarding mechanism. If, on the other hand, a packet without custom data, it is subjected to the forwarding mechanism indicated in the previous approach.

#### **P4 program for defining the packet-processing pipeline in the second strategy:**

The given P4 program acts for configuring the packet-processing pathway in a programmable switch, particularly when the switch recirculates customized packets cyclically. This program defines the procedures for handling incoming network packets, including recirculation and forwarding, all of which are dependent on the header characteristics. Let's explain the key features of this program in further detail:

- **Headers and metadata:**

In this implemented strategy, the packet headers closely resemble those of the previous approach, with one key distinction: the introduction of an additional header called "customData". This customData header comprises four fields: `proto_id`, `content_id`, `ingress_num`, and `egress_num`. The `content_id` field determines the type of custom data, typically represented as a numerical value that defines variations among custom packets. `Ingress_num` signifies how many times a packet enters the egress pipeline, while `egress_num` indicates the number of times it exits from the egress pipeline. The `proto_id` field is employed by the parser to establish a transition state for custom packets. All other headers, such as IPv4 and Ethernet, as well as metadata like recirculation counter, are identical to those utilized in the previous strategy. You can see the complete code for this part in [Figure 4.9](#)

- **Parser and Deparser:**

The strategy recirculation-based method for custom packets, as previously explained, provides a separation in the handling of the `etherType` field within the Ethernet header. There are two separate options: `TYPE_IPV4`, which refers to ordinary packets and initiates a parser process similar to the previous technique, and `TYPE_CUSTOMDATA`, which refers to custom packets. When the `etherType` assumes the value of `TYPE_CUSTOMDATA`, the finite state machine governing the parser transitions into the "parse\_customData" state. Within this state, the parser extracts the customData header, and its subsequent action dependent on the "proto\_id" field within this header. If the "proto\_id" matches `TYPE_IPV4`, the parser selects the "parse\_ipv4" state for further processing.

It's critical to note that "TYPE\_IPV4" and "TYPE\_CUSTOMDATA" represent constant values predefined in the initial lines of the code. The parser references these values embedded in the headers to determine the subsequent state to transition to. (see [Figure 4.9](#))

In the deparser component, the only noticeable difference between this method and the previous one is the necessity of attaching the customData header to the packet payload. A comprehensive code representation of the parser and deparser components for this strategy is available in [Figure 4.10](#) for reference.

```

const bit<16> TYPE_IPV4 = 0x800;
#define RECIRCULATE_TIMES 3
const bit<16> TYPE_CUSTOMDATA = 0x1313;
/***** H E A D E R S *****/

typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

/* TCP/IP Headers */

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> tos;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header customdata_t {
    bit<16> proto_id;
    bit<16> content_id;
    bit<8> ingress_num;
    bit<8> egress_num;
}

struct metadata {
    @field_list(1)
    bit<8> counter;
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    customdata_t customdata;
}

```

Figure 4.9. Defining headers in the p4 file for the strategy that custom packets are recirculated

- **Ingress processing:**

The ingress pipeline acts exactly as the previous strategy and is responsible for packet forwarding.

- **Egress processing:**

In this strategy, the egress pipeline focuses on packet recirculation within the switch like the previous strategy. The MyEgress control block handles recirculation decisions. If the packet's recirculation counter is below a predetermined threshold (RECIRCULATE\_TIMES), the packet is recirculated within the switch.

```

/***** P A R S E R *****/
parser MyParser(packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            TYPE_IPV4: parse_ipv4;
            TYPE_CUSTOMDATA: parse_customdata;
            default: accept;
        }
    }

    state parse_customdata {
        packet.extract(hdr.customdata);
        transition select(hdr.customdata.proto_id){
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}

/***** D E P A R S E R *****/
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.customdata);
        packet.emit(hdr.ipv4);
    }
}

```

Figure 4.10. This figure illustrates the parser and deparser components of the recirculation-based strategy designed for custom packets. The figure highlights the custom packet handling, etherType field differentiation, state transitions, and key processing steps.

In the egress pipeline, we have implemented a set of actions specifically tailored for custom packets. These actions are essential for managing custom packet data as it flows through the pipeline.

**Update\_CustomData Action:** This action is responsible for updating two vital fields within the custom data header: `ingress_num` and `egress_num`. Each time a custom packet enters and exits the egress pipeline, these fields are automatically incremented by one. This mechanism tracks the passage of custom packets through the pipeline.

**CustomData\_Forward Action:** The role of this action is straightforward yet crucial – forwarding custom packets to their intended destinations. By specifying the egress port of the switch, this action ensures custom packets reach their designated endpoints. The destination of custom packets is determined based on their `content_id`, ensuring precise routing.

**Recirculation Action:** When invoked, this action initiates the recirculation process. Essentially, it allows the egress pipeline to recirculate the packet within the switch one additional time.

For a comprehensive view of these actions and their implementation, please refer to Figure 4.11.

```

***** EGRESS PROCESSING *****
*****/

control MyEgress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        // Define the output port
        standard_metadata.egress_spec = port;
        // Update src and dst MACs according to the current switch
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        // Decrease TTL by one when forwarding the packet
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    action update_customdata_processing_count_by_num(in bit<8> count_num){
        hdr.customdata.ingress_num = hdr.customdata.ingress_num + count_num;
        hdr.customdata.egress_num = hdr.customdata.egress_num + count_num;
    }

    action customdata_forward(egressSpec_t port){
        standard_metadata.egress_spec = port;
    }

    action recirculate_packet(){
        recirculate_preserving_field_list(1);
    }
}

```

Figure 4.11. This figure presents an overview of the actions defined in the egress pipeline for custom packets. It highlights the roles and functionality of each action, including updating custom data fields, forwarding custom packets, and initiating recirculation. These actions collectively facilitate the efficient handling and routing of custom packets within the switch's egress pipeline.

The configuration of the egress pipeline necessitates the definition of tables, matches, and actions to govern the processing of custom packets. To accomplish this, we begin by creating a forwarding table where the match condition is based on the `content_id` of custom packets. The associated action for these matches is the `customdata_forward` action.

Once the table and actions are defined, we need to specify when and how these match-action pairs apply to incoming packets. This determination depends on whether a packet is recognized as "custom". This is determined by inspecting the packet's header. Specifically, if the header contains the custom data portion, we classify the packet as custom, and the egress pipeline proceeds accordingly.

The conditions for applying actions to custom packets are as follows:

**Recirculation:** When a packet is categorized as custom, it undergoes recirculation

within the egress pipeline if the count of times it has entered the egress pipeline is less than a predefined value (as defined earlier).

**Forwarding:** After the specified number of recirculations, the packet is forwarded based on the type of custom packet it represents, determined by its `content_id`. The associated `customdata_forward` action is executed to route the packet appropriately.

**Update\_CustomData:** Upon entry into the egress pipeline, the packet’s counters for entering and exiting the pipeline are updated using the `update_customdata` action.

For a detailed view of the rules, matches, and actions governing custom packet processing within the egress pipeline, please refer to Figure 4.12.

```
//To debug
table recirculate {
    key = {
        meta.counter: exact;
    }
    actions = {
        NoAction;
    }
    size=1;
    default_action=NoAction();
}

table customdata_forward_table{
    key = {
        hdr.customdata.content_id: exact;
    }
    actions = {
        customdata_forward;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}

apply {
    /* recirculate.apply();
    if (meta.counter < RECIRCULATE_TIMES){
        meta.counter = meta.counter + 1;
        recirculate_preserving_field_list(1);*/
    if (hdr.customdata.isValid()) {
        update_customdata_processing_count_by_num(0x00000001);
        // if (standard_metadata.instance_type != PKT_INSTANCE_TYPE_INGRESS_RECIRC) {
        if (hdr.customdata.ingress_num != PKT_INSTANCE_TYPE_INGRESS_RECIRC) {

            recirculate_packet();
        }
        customdata_forward_table.apply();
    }
}
```

Figure 4.12. This figure provides insight into the critical components of the egress pipeline configuration for custom packets. It illustrates the forwarding table, match conditions based on `content_id`, and the corresponding actions, including recirculation, forwarding, and updating custom data counters. These elements collectively define the behavior of custom packets as they traverse the egress pipeline.

### Switch Controller:

In terms of the controller’s role in this strategy, the pattern parallels the previous approach. For normal packets, the controller instructs the switch to execute matching based

on the packet's destination IP address. This results in the switch's action, which involves the provision of two parameters: the MAC destination address and the port number for forwarding. Conversely, for custom packets, the controller mandates the `content_id` of the custom packet as the matching field, directing the switch to execute actions tailored to the packet's unique requirements. Additionally, the controller specifies the port number for forwarding the custom packet to its designated destination. In terms of differentiating custom packets within this second strategy, an essential distinction is made based on their intended destination. Consider a custom packet with the `content_id` of 1. The specified destination IP address in this situation is `x.x.x.x`. As a result, the coordinated forwarding of this unique packet entails routing it through switch port one. When a custom packet with a `content_id` of 2 is encountered, the destination IP address converts to `y.y.y.y`. Similarly, the packet's path is routed through the switch's port 2. This fine distinction in forwarding techniques is precisely designed to distinguish unique packets from standard ones, adjusting how they move through the network to their different characteristics. This precisely designed technique guarantees that each custom packet is treated specifically, with its path corresponding to its `content_id`. This solution provides an extra layer of control, allowing the programmable switch to respond to different packet types dynamically, improving the overall efficiency and usefulness of the recirculation algorithm. You can see the code for the controller in strategy 2 in [Figure 4.13](#).

## 4.2 How to Control Delay?

After discussing the development of the packet recirculation algorithm, let's look at how it may be employed to control packet delay in a network. As previously explained, this approach includes defining a constant value to determine how many recirculation cycles the switch should do for a given packet. The next stage will be to investigate how this recirculation may be used to impact packet delay. The primary question in our approach is about the change in packet latency with each successive recirculation. Understanding this connection is critical because it allows us to determine how many recirculations are necessary to attain a given delay threshold, such as `X ms`. However, directly measuring the delay experienced by a packet with each recirculation is a complex task. It's not deterministic, requiring an average over multiple packets to obtain a reliable result. To overcome this challenge, a more realistic approach is to observe the delay from the perspective of a source host producing packets, which are then transferred to a receiving destination host. Our programmable switch sits in the midst of this transmission. Initially, we configure the switch for zero recirculations, which allows us to determine the initial delay experienced by packets as they traverse the network. By analyzing the outcomes of this experiment, we can approximate the aggregate delay arising from host processing, network propagation, and transmission delays. This baseline serves as a reference including the overall latency introduced by network devices, links, and processing. Subsequently, we increment the recirculation count and repeat the experiment, capturing the delay under various recirculation settings. By comparing the new delay values against the baseline, we can isolate the additional delay attributed specifically to the switch due to recirculation. This information is instrumental in comprehending the

impact of recirculation on delay and allows us to exercise fine control over network performance. The classic tool known as Ping comes in assistance when it comes to monitoring packet latency inside a network. We can easily measure the time it takes for a packet to transit from a source host to a destination host using the Ping program. This time interval includes a spectrum of latency contributors, including host processing, network propagation, transmission via links, and the switch's involvement in recirculation. To conduct this evaluation, we construct a network topology comprising a source host, a destination host, and a programmable BMv2 switch. Our procedure involves configuring the desired number of recirculations within the switch and subsequently performing Ping experiments. By analyzing the delay experienced by ICMP packets corresponding to varying recirculation counts, we can figure out the influence of recirculation on network delay. Moreover, we can introduce variations in link bitrates and packet sizes to analyze their impact on our measurements. The manipulation of link bitrates can be facilitated through a traffic controller tool while adjusting ICMP packet sizes is achievable through the customization options provided by the Ping utility. This comprehensive approach ensures a robust evaluation of delay caused by recirculation while accounting for various network parameters.

You can find the results of the ping experiment in Figure 4.14, which specifically captures the scenario where the packet size is set at 1200 bytes, and where no specific number of recirculations has been defined within the programmable switch.

In the ping command used for these experiments, several key options were instrumental in shaping the outcomes. The first of these is `-s`, which allows us to specify the number of data bytes to be sent. By default, this value is set at 56, translating to 64 ICMP data bytes when combined with the 8 bytes of ICMP header data. The `-c` option plays a crucial role by determining the count of ICMP packets. It dictates when the ping command should stop after sending a set number of `ECHO_REQUEST` packets. The `-i` option, on the other hand, is responsible for setting the interval between sending each packet. Normally, the default is to wait for one second between each packet, or not to wait at all in flood mode. However, it's worth noting that only a super-user can set the interval to values less than 0.2 seconds. Our experimental approach maintained a consistent packet size of 1200 bytes, with variations in the other options and ping command instructions tailored to the specific needs of each experiment.

```

"target": "bmv2",
"p4info": "build/recirculation.p4.p4info.txt",
"bmv2_json": "build/recirculation.json",
"table_entries": [
  {
    "table": "MyIngress.ipv4_lpm",
    "default_action": true,
    "action_name": "MyIngress.drop",
    "action_params": { }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:01:11",
      "port": 1
    }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:02:22",
      "port": 2
    }
  },
  {
    "table": "MyEgress.customdata_forward_table",
    "match": {
      "hdr.customdata.content_id": [101]
    },
    "action_name": "MyEgress.customdata_forward",
    "action_params": {
      "port": 1
    }
  },
  {
    "table": "MyEgress.customdata_forward_table",
    "match": {
      "hdr.customdata.content_id": [102]
    },
    "action_name": "MyEgress.customdata_forward",
    "action_params": {
      "port": 2
    }
  }
]

```

Figure 4.13. The Controller of the switch in the state that the switch recirculates specific packets, the controller indicates extra match-actions for recirculating the packets. This figure illustrates the controller's essential role in Strategy 2 where the switch recirculates custom packets, showcasing the differentiated handling of custom packets based on their content\_id and destination IP address.



```
PING 10.0.2.2 (10.0.2.2) 1200(1228) bytes of data.  
1208 bytes from 10.0.2.2: icmp_seq=1 ttl=63 time=1.37 ms  
1208 bytes from 10.0.2.2: icmp_seq=2 ttl=63 time=1.18 ms  
1208 bytes from 10.0.2.2: icmp_seq=3 ttl=63 time=1.15 ms  
1208 bytes from 10.0.2.2: icmp_seq=4 ttl=63 time=1.88 ms  
  
--- 10.0.2.2 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3004ms  
rtt min/avg/max/mdev = 1.150/1.393/1.881/0.293 ms
```

Figure 4.14. Results of the ping experiment with a packet size of 1200 bytes and no specific number of recirculations defined in the programmable switch.



## Chapter 5

# Experimental/numerical evaluation

### 5.1 Methodology

In this section, I present the methodology used to investigate the behavior and performance of programmable switches in the context of Software-Defined Networking (SDN). I aimed to gather significant information on the influence of recirculation in programmable switches on network packet delay and throughput by creating a complete network setup and carefully selecting measurement tools. We chose an Ubuntu virtual machine (VM) as our platform of choice for the tests and then repeated the experiments on a real Linux server. An Ubuntu VM provides a stable and self-contained environment in which we may run the complete instance of the Ubuntu operating system in a virtualized environment. Ubuntu, a famous open-source Linux distribution, has an array of features and capabilities, making it an excellent choice for network simulation and testing. Moreover, within the Linux environment, we used the power of Mininet, a highly capable network emulator. With Mininet, we constructed and emulated various network topologies.

To emulate the network environment for our experiments, we developed the following key files: BMv2 P4 Switch Configuration File that contains the P4 code that defines how the BMv2 switch processes incoming packets and then the controller script that is responsible for defining the high-level network policies and commanding the switch on where to send the incoming packets based on their headers. I explained completely the development of instructions for the programmable switch with p4, the way that the controller commands the switch on our approach and the code of these files in the previous chapter.

The third part that is essential for simulating our network to see the behavior of programmable switches in the network and how they can control the delay of packets with recirculation inside the programmable switch is to define a network topology and make an appropriate network. Mininet requires a topology definition file written in Python (e.g., `topology.py`) to describe the network layout. In this script, we utilized Mininet's Python API to define the topology, specifying the hosts, switches, and links.

### 5.1.1 Network topology overview:

In our network setup, we have two main computers, one acting as the source and the other as the destination. These computers want to exchange data, and in between them, we have a BMv2 switch that helps manage this communication. You can see the topology in the Figure 5.1

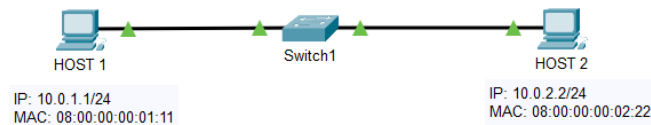


Figure 5.1. The network topology configuration when in the network there is one programmable switch

- **Hosts and their interfaces:**

Each of the two computers (hosts) has two ways to communicate. One is through their internal communication loop, called the loopback interface. The second is the interface that connects them to the switch. This interface helps them send data packets to the switch, which will then route them to the appropriate destination. Each of these interfaces has an IP address and a MAC address. In Figure 5.2 you can see how these configurations are set for the hosts in the topology file.

```

"hosts":{
  "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
    "commands":["route add default gw 10.0.1.10 dev eth0",
      "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
  "h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
    "commands":["route add default gw 10.0.2.20 dev eth0",
      "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]}
},
  
```

Figure 5.2. The configuration of hosts in topology file

- **BMv2 Switch and its ports:**

The BMv2 switch is like a traffic manager. One of its ports connects to the source host, while the other connects to the destination host. This switch is programmable, meaning we can control how it handles data traffic.

- **Link connections and parameters:**

There are two important links in our setup. One link connects the source host to the switch, and the other connects the switch to the destination host (as you can see in Figure 5.3). These links determine how fast data can travel (link bitrate) and how long it takes for data to move from one end to the other (propagation delay). We can change these parameters in our experiments to see how they affect the network's performance.

```

"switches":{
  "s1": { "runtime_json" : "pod-topo/s1-runtime.json"
},
"links": [
  ["h1", "s1-p1"], ["h2", "s1-p2"]
]

```

Figure 5.3. Set the Controller of the switch and connecting ports with links together in the topology file

### 5.1.2 Extended Topology with Multiple Switches:

We expanded our network architecture to include two BMv2 switches after understanding the single-switch examinations(see Figure 5.4), thereby creating a more intricate setup. In this extended configuration, the interplay between multiple switches introduces a new layer of complexity to the network's dynamics. Specifically, the source host is now connected to the first switch, and this switch is further linked to the second switch. The second switch, in turn, connects to the destination host. This multi-switch arrangement allows us to explore the behavior of recirculation when data packets traverse multiple programmable switches.

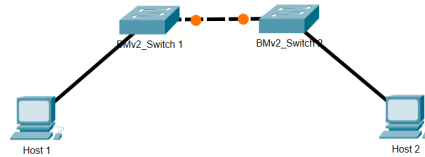


Figure 5.4. The network topology configuration when in the network there are two programmable switches

## 5.2 Numerical results

After providing the structure for packet recirculation algorithm development and implementation within programmable switches, the focus of this part shifts to evaluating the

real-world impact of this new technique. The preceding sections have shown the mechanisms and strategies associated with packet recirculation, demonstrating its ability to control packet delay and its effect on network performance. The evaluation step now seeks to experimentally evaluate the results of these theoretical parts through actual experiments.

In this section, we examine the efficiency of the packet recirculation method in controlling packet delays as well as its wider consequences on network performance. We investigate how the recirculation process interacts with network dynamics, including link parameters, traffic patterns, and the varied features of distinct data streams. We want to provide helpful information about the level of control available over packet delays and their impact on total network throughput by carefully analyzing the outcomes of these tests.

The evaluation process comprises a series of interconnected steps, each meticulously designed to explore different dimensions of the recirculation approach. After an overview of the experimental environment and methodology in the previous section, I present actual data from multiple tests done both in a controlled virtual machine and on a real server in this section. This strategic duality enables us to demonstrate the relevance of discoveries across the gap between virtual abstraction and real one. This becomes critical in evaluating the practical feasibility of our suggested technique. This comprehensive method clarifies differences in processing times, transmission latencies, and other pertinent factors, ultimately enriching our understanding of the difference between virtual and actual scenarios.

As we navigate through the presentation, analysis, and discussion of results, we highlight the trends, deviations, and implications discovered during the evaluation. Furthermore, we acknowledge the inherent limitations of our experiments, including factors that could introduce variability into the outcomes. By critically interpreting our findings, we gain an understanding of the interplay between packet recirculation, delay of packets, and broader network performance metrics.

Ultimately, the evaluation section aims to bridge the gap between theoretical constructs and their practical effects, thereby solidifying the thesis's contribution to the understanding and advancement of programmable data plane techniques. Through this scientific journey, we hope to provide network administrators, researchers, and developers with an actionable understanding of controlling the delay of packets with programmable switches.

### **5.2.1 Analyzing Packet Delay and Processing Times in a Recirculation-Free Environment**

In this initial segment of the results, we go into the fundamental assessment of packet delay without the influence of recirculation within the BMv2 switch. This investigation aims to reveal the intrinsic processing and transmission times associated with the switch and host components, under varying link bitrates. Furthermore, a precise comparison is established between the theoretically projected outcomes which is explained later in this section and the experimental results extracted from our conducted experiments. Additionally, an analysis examines the variances between executing these experiments in the virtual

environment and the real-world context. This distinction enables us to properly measure changes in processing and transmission times within these various operational areas.

To begin our investigation, we will investigate the measurement of packet delay in a topology consisting of a source host, a destination host, and a connecting BMv2 switch. Our methodology of delay measurement employs the PING tool, which provides us the Round-Trip Time (RTT), signifying the duration for an ICMP echo request to traverse from the source host to the destination host and return as an echo reply. This RTT includes various components such as processing times within devices, transmission durations, queuing delays, and propagation periods within the network.

For our investigation, we start by estimating the theoretical RTT values before conducting experiments. By performing the experiments both in a virtual environment and on a real Linux server, we aim to observe disparities between the experimental and theoretical RTT values. To comprehend these disparities, it's essential to outline the participants of the theoretical RTT.

The propagation delay, indicating the time for a signal to traverse between network points, as well as the queuing delay, denoting the time a packet waits in network queues before transmission, can be considered negligible for our purposes. This is attributable to our topology's simplicity and the co-location of virtual hosts on the same physical machine, eliminating the practical significance of distance. Furthermore, the low network load and immediate processing within the switch mitigate the possibility of congestion, warranting a negligible queuing delay.

The remaining contributing factors to RTT are the processing and transmission delays. The transmission delay relies on both the link bandwidth connecting the sender and receiver, as well as the data packet's size. The calculation for transmission delay follows the formula:  $\text{Transmission Delay} = \text{Packet Size} / \text{Bandwidth}$ , where Packet Size denotes the data packet size in bits and Bandwidth signifies the channel capacity in bits per second (bps). for our experiments, we set the packet size equal to 1200 bytes and firstly, we do the experiments without specifying any link bitrate so the link bitrate would be the maximum that our system can handle the traffic and the transmission delay would be negligible. we change the link bitrate to 1Gbps, 1Mbps, 100Kbps, and 10Kbps and the transmission delay for these link bitrates theoretically would be:

$\text{Transmission Delay} = \text{Packet Size(bits)} / \text{Bandwidth(bps)}$

NO linkbitrate:  $\text{Transmission delay} = 1228 * 8 / \infty \approx 0$

Link bitrate = 1Gbit/s:  $\text{Transmission delay} = 1228 * 8 / 10^9 = 0.000009824 \text{ s} = 0.009824 \text{ ms}$

Link bitrate = 1Mbit/s:  $\text{Transmission delay} = 1228 * 8 / 10^6 = 0.009824 \text{ s} = 9.824 \text{ ms}$

Link bitrate = 100Kbit/s:  $\text{Transmission delay} = 1228 * 8 / 10^5 = 0.09824 \text{ s} = 98.24 \text{ ms}$

Link bitrate = 10Kbit/s:  $\text{Transmission delay} = 1228 * 8 / 10^4 = 0.9824 \text{ s} = 982.4 \text{ ms}$

We consider the packet size 1228 bytes because to the ICMP packets that are sent in the network IP header (20Bytes) and Ethernet header (8Bytes) are added.

To approximate the processing delay, a series of ping experiments were executed without specifying link bitrates. The observed minimum time across these 1000 experiments provided an effective estimation of the processing time incurred by both hosts and the

switch because, with these settings in our experiment, we can also remove the transmission delay and make it negligible, and the only component that remains is the processing delay. By choosing the minimum, we can consider it the processing delay of our components; other processing delays, for example, that are caused by the virtual environment or CPU, have no role in it.

Finally, for estimating RTT, I sum these components for different link bitrates, and the results would be:

RTT = Transmission delay + Processing delay + Queuing delay + Propagation delay

Queuing delay = Propagation delay = 0

NO link bitrate: RTT = 0.98ms  $\rightarrow$  Processing delay = 0.98ms

Link bitrate = 1Gbit/s: RTT =  $0.009824 * 4 + 0.98 = 1.019$ ms

Link Bitrate = 1Mbit/s: RTT =  $9.824 * 4 + 0.98 = 40.276$ ms

Link Bitrate = 100Kbit/s: RTT =  $98.24 * 4 + 0.98 = 393.94$ ms

Link Bitrate = 10Kbit/s: RTT =  $4 * 982.4 + 0.98 = 3930.58$ ms

For computing the RTT in different link bitrates, we multiply, the transmission delay by 4 because the packets cross 4 links to reach the destination and return to the source host.

This comprehensive analysis affords us a foundational understanding of the theoretical aspects involved in RTT, enabling a comparison between these theoretical values and real-world experimentation. Subsequently, the comparison between virtual and real environments highlights the small variations in processing times and transmission durations present in each scenario.

After these computations, as our theoretical values for RTT with different link bitrates, it is now time to do the experiments on a virtual Linux machine and a real Linux server. In these experiments, I conducted a series of "ping" tests using the command "ping 10.0.2.2 -s 1200 -c 1000 -i 0.1". In this command, the source client sends an ICMP echo request to the destination host, whose IP address is indicated in the command (10.0.2.2), and -s, -c, and -i are the options of the ping tool. -s indicates the size of ICMP packets that would be sent to the destination, -c indicates that on the source host, 1000 packets would be produced and sent, and the -i option indicates that the time period between sending these packets to the destination would be 0.1 second. To modify the link bitrates, I employed the "tc qdisc change dev ethX root netem rate Xbit" command across all four links. In figure 5.5, you can see the results of these experiments and the comparison between the minimum and average RTT in these experiments for different link bitrates with the theoretical RTT values for different link bitrates that I computed previously.

I have done all of these steps on the Linux server to see how different the results would be in a virtual environment and a real one. The difference that we should notice is that here, when we do the ping experiments without specifying any link bitrates to get the processing time of hosts and the BMv2 switch, it would be different, and it would change from 0.98 ms to 2.16 ms, and consequently, we should compute the theoretical values for RTT again as follows: RTT = Transmission delay + Processing delay + Queuing delay + Propagation delay

Queuing delay = Propagation delay = 0

NO link bitrate: Transmission delay =  $1228 * 8 / \infty \approx 0 \rightarrow$  RTT = 2.16ms  $\rightarrow$  Processing delay = 2.16ms



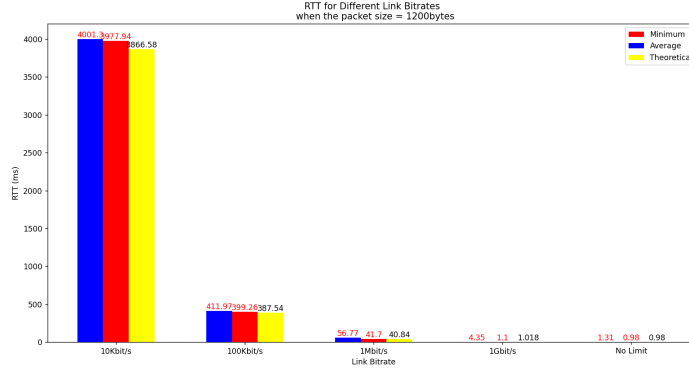


Figure 5.5. Theoretical RTT vs. minimum and average RTT in Ping experiments when the packet size is 1200 bytes for the packets with no recirculation in the BMv2 switch with different link bitrates in the virtual machine

Link bitrate = 1Gbit/s: Transmission delay =  $1228 * 8 / 10^9 = 0.000009824s = 0.009824ms$   
 $\rightarrow RTT = 0.009824 * 4 + 2.16ms = 2.199ms$

Link Bitrate = 1Mbit/s: Transmission delay =  $1228 * 8 / 10^6 = 0.009824s = 9.824ms$   
 $\rightarrow RTT = 9.824 * 4 + 2.16 = 41.456ms$

Link Bitrate = 100Kbit/s: Transmission delay =  $1228 * 8 / 10^5 = 0.09824s = 98.24ms$   
 $\rightarrow RTT = 98.24 * 4 + 2.16 = 395.12ms$

Link Bitrate = 10Kbit/s: Transmission delay =  $1228 * 8 / 10^4 = 0.9824s = 982.4ms$   
 $\rightarrow RTT = 4 * 982.4 + 2.16 = 3931.76ms$

and then do experiments exactly like the ones that I have done before in the virtual machine. In the figure 5.6 you can see the experiments in this environment and the comparison between the minimum and average RTT in the Ping experiments on the Linux server vs. the theoretical values that I computed. In the figure, 5.7 you can see the comparison between the minimum and average RTT in Ping experiments that have been done in the virtual machine for different link bitrates vs. the minimum and average RTT in Ping experiments that have been done on the Linux server for different link bitrates.

Now it is the time for analyzing these experiments and comparisons. Upon analyzing the graphs, notable trends emerge regarding the processing times of components under different conditions. Firstly, it becomes evident that there is a noticeable increase in processing time when conducting the ping experiments within the Linux server environment. In the virtual environment, the processing time averages at 0.98 ms, whereas in the Linux server, this figure notably rises to 2.16 ms.

Furthermore, a surprising correlation between link bitrate and processing delay occurs. As the link bitrate decreases, resulting in increased transmission delay, there is a corresponding increase in processing delay within the components. This observation suggests a potential relationship between lower link bitrates and elevated overall processing times. This phenomenon could be attributed to several factors, including increased processing demands on the components to handle packets and transmit them through the

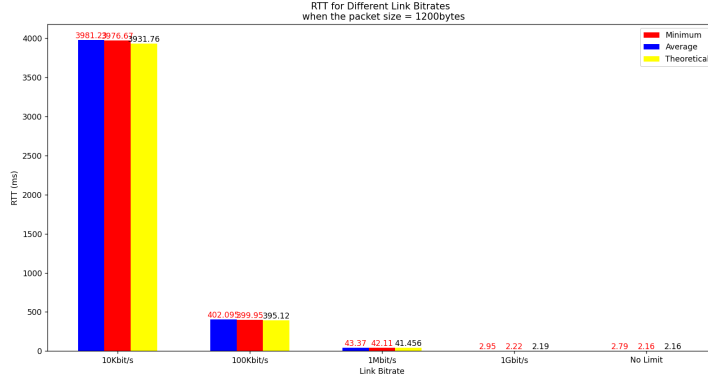


Figure 5.6. theoretical RTT vs. minimum and average experimental RTT in Ping experiments when the packet size is 1200 bytes for the packets with no recirculation in the BMv2 switch with different link bitrates on the Linux server

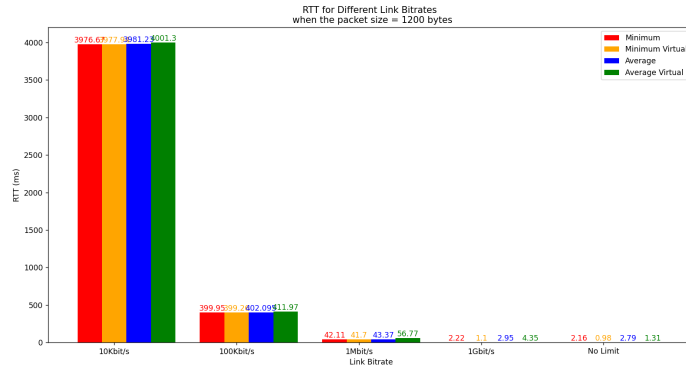


Figure 5.7. minimum and average experimental RTT on the Linux virtual machine vs. minimum and average experimental RTT on the Linux server in Ping experiments when the packet size is 1200 bytes for the packets with no recirculation in the BMv2 switch with different link bitrates

links to their destination. Additionally, it's conceivable that the processing times of the CPU, operating system, Mininet, or the virtual machine itself in the virtual environment may experience increases when link bitrates decrease, contributing to the observed rise in overall processing times.

Furthermore, a noteworthy distinction arises when comparing the experiments conducted in the Linux virtual machine with those in the Linux server. In the Linux server environment, it becomes apparent that the difference between the minimum ping time and the average time is remarkably low, with these values closely aligning. Conversely, in the virtual environment, a noticeable disparity exists between these metrics.

This distinction is primarily attributable to resource limitations in the virtual environment. Within this constrained setting, the processing times for many packets are notably influenced by virtualization factors such as resource allocation, virtual memory, and virtual CPU utilization. Consequently, this leads to a significant increase in the Round-Trip Time (RTT) for numerous packets, creating a substantial gap between the minimum RTT and the average RTT. These findings underscore the substantial impact of resource constraints on processing times within virtualized environments, which in turn can influence network performance.

### 5.2.2 Investigating the Impact of Recirculation on Packet Delay in BMv2 Switches

The subsequent phase of our investigation is centered on exploring the impacts of changing the number of recirculations within the BMv2 switch on packet delay. This important part of our study offers a valuable understanding of the relationship between recirculation instances and packet delays within the programmable switch.

Just as in the previous experiments, this set of trials involves the manipulation of link bitrates to comprehend the interplay between recirculation instances and varying network conditions. By varying the number of packet recirculations -namely 0, 1, 3, 7, and 15- we aim to capture the varying delay patterns introduced as packets traverse through the switch, with the number of transmissions serving as our metric. It should be noted that the number of transmissions is always one greater than the number of recirculations, showing the total number of times a packet crosses the switch. This progression follows the order of "1, 2, 4, 8, and 16," wherein each subsequent transmission is twice the previous value. This specific ordering facilitates numerical analysis, which will be elaborated on in the subsequent result analysis.

It's essential to emphasize that the number of recirculations directly influences the complexity of the packet's route within the switch, impacting its overall delay. For instance, when the number of recirculations is set to zero, the packet traverses the switch only once, translating to a single transmission with a negligible delay. As we increment the recirculation count, the number of transmissions grows, providing us with a comprehensive understanding of how recirculation instances impact the delay in a multipass scenario.

This segment of our analysis involves a comparative study of results obtained from experiments conducted within both a Linux virtual machine environment and an actual Linux server. The contrast between these two settings will provide an understanding of the potential variations in processing times and transmission times between the virtual and real environments. Within these experiments, we focus on extracting two key metrics: the minimum Round-Trip Time (RTT) from a series of 1000 Ping tests for each recirculation count and link bitrate, as well as the average RTT derived from these experiments.

Now let's see the graphs. I put the results of each experiment for each link bitrate both in the virtual environment and on the Linux server, one beside the other, to make the comparison easier. The first graph [5.8](#) is for the ping experiment in the virtual environment, and the second one [5.9](#) is for the same experiment on the Linux server

when I did not specify any bitrate for all the links. In these experiments, like before, the packet size is 1200 bytes, and we have done the Ping experiment 1000 times. In these experiments, you can see how the delay of the packets in the network from source to destination, which I indicate as the RTT of packets (y-axis), changes with varying the number of transmissions in the BMv2 switch (x-axis).

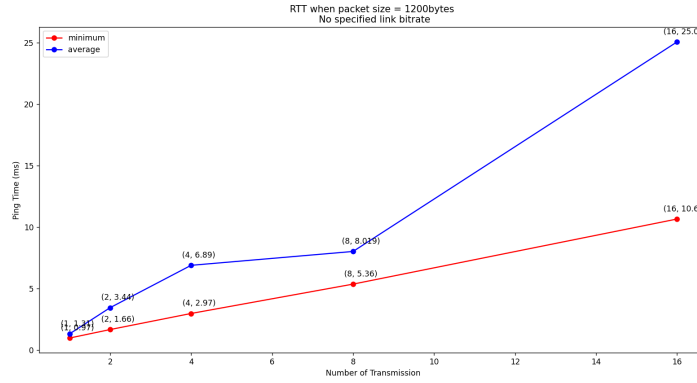


Figure 5.8. Minimum and average experimental RTT in the Linux virtual machine in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch without specifying any link bitrate

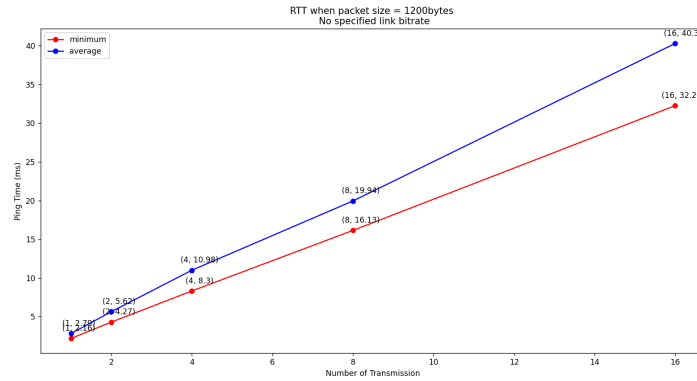


Figure 5.9. Minimum and average experimental RTT in Linux Server in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch without specifying any link bitrate

**Analyse:** A closer inspection of the new data reveals more about the link between packet delay and the number of recirculations within the BMv2 switch. Let's get started with the analysis by looking at the outcomes acquired in the virtual environment.

In the initial scenario, when there is no recirculation within the BMv2 switch, the

minimum Round-Trip Time (RTT) is recorded at 0.97 ms. This delay primarily stems from the processing times incurred by network components, including the hosts and the BMv2 switch. However, as the number of transmissions increases to 2, indicating a single recirculation in the BMv2 switch, the minimum RTT experiences a noticeable increase to 1.66 ms. This change has an additional impact on the delay: 0.98 ms attributed to component processing and 0.68 ms associated with the recirculation process. Formulating this relationship, we arrive at a short equation:  $\text{delay} = 0.98 + 0.68 * X$ , where  $X$  represents the number of recirculations.

For example, when  $X$  equals 3 for three recirculations, the predicted delay is 3.02 ms. Remarkably, the experimental results align closely with this theoretical projection, demonstrating a delay of 2.97 ms. Likewise, for seven recirculations, which correspond to eight transmissions, the theoretical delay amounts to 5.74 ms, while the experimental results yield a value of 5.36 ms. This close concordance between theory and practice provides valuable guidance on the number of recirculations required for effective packet delay control.

Notably, these observations also extend to the experiments conducted in the Linux server environment. In this context, when there is no recirculation, the delay attributed to component processing is approximately 2.16 ms. However, with one recirculation in the BMv2 switch, this delay increases to 4.27 ms. The corresponding theoretical relationship can be expressed as:  $\text{delay} = 2.16 + 2.11 * X$ , where  $X$  still signifies the number of recirculations.

For instance, with  $X$  set at 3 for three recirculations, the theoretical delay is 8.49 ms, closely paralleling the experimental result of 8.3 ms. Similarly, for seven recirculations, the theoretical delay equates to 16.93 ms, aligning remarkably well with the experimental outcome of 16.13 ms.

Furthermore, the processing delay caused by recirculation in the BMv2 switch in the Linux server environment is significantly larger, measuring 2.11 ms against 0.68 ms in the virtual environment. The difference highlights the significant influence of the computing environment on processing times, demonstrating the complexities of network performance in different settings.

The next two graphs show the results when changing the link bitrates to 1 Gbps. The first graph [5.10](#) shows the results when doing the experiments in the virtual Linux environment, and the second one [5.11](#) shows the results when doing the experiments on the Linux server.

**Analyse:** This group of graphs shows the Round-Trip Time (RTT) results when the link's bitrate is set to 1 Gbps in both the virtual machine and Linux server settings. These findings enable us to make meaningful comparisons about the influence of network settings on packet delay control.

Close inspection reveals that the minimum RTT outcomes in these graphs closely match the outcomes of our previous investigations, in which we did not specify a link rate. Any differences between these results can be due to the tiny transmission delays caused by the reduced link bitrate, a reasonable effect that matches our expectations.

When we look at the average RTT findings in the Linux server environment, we see a substantial difference. While it is very normal for our studies to show non-deterministic

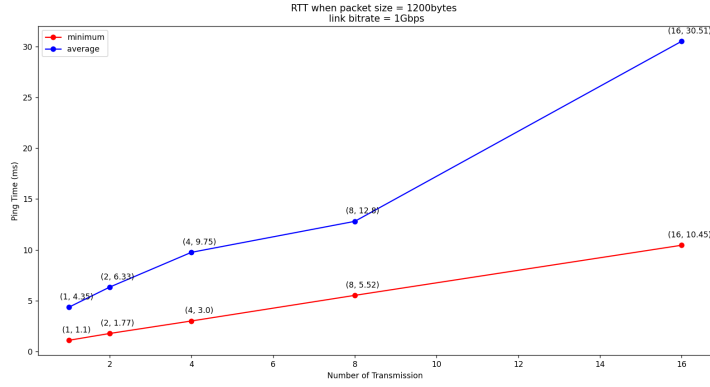


Figure 5.10. Minimum and average experimental RTT in the virtual Linux machine in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 1Gbps

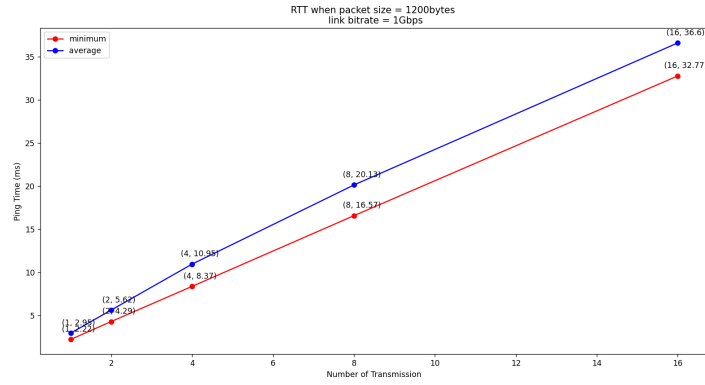


Figure 5.11. Minimum and average experimental RTT in Linux Server in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 1Gbps

behavior due to the wide range of factors influencing network latency, an interesting pattern occurs. In the Linux server environment, the average RTT values, like the minimum RTT, have a steady and linear relationship with the number of recirculations. As the number of recirculations grows, so does the average RTT, indicating that the behavior follows a predictable pattern.

In the virtual environment, however, a clear difference appears. In the virtual machine environment, the average RTT figures do not nearly match the minimum RTT findings. The difference between the average and minimum RTT is unpredictable at each point on the graphs. This irregular behavior highlights the significant impact of resource constraints in the virtual environment. Within the virtual machine, where resources like

CPU and memory are limited, these constraints might cause unforeseen packet processing delays. This phenomenon is not limited to the 1 Gbps connection bitrate studies; it occurs in all instances in which PING experiments were carried out within the virtual machine.

In summary, these findings emphasize the effects of computational resources on network performance, particularly in virtualized environments. While the Linux server environment exhibits stable and predictable behavior in terms of RTT, the virtual machine environment demonstrates unpredictable fluctuations in average RTT due to resource limitations.

The following two graphs show the results when changing the link bitrates to 1 Mbps. The first graph 5.12 shows the results when doing the experiments in the virtual Linux environment, and the second one 5.13 shows the results when doing the experiments on the Linux server.

The next two graphs show the results when changing the link bitrates to 100 kbps. The first graph 5.14 shows the results when doing the experiments in the virtual Linux environment, and the second one 5.15 shows the results when doing the experiments on the Linux server.

The next two graphs show the results when changing the link bitrates to 10 kbps. The first graph 5.16 shows the results when doing the experiments in the virtual Linux environment, and the second one 5.17 shows the results when doing the experiments on the Linux server.

**Analyze:** In addition to the previous analysis, it's worth emphasizing the effect of decreasing link bitrates on packet delay, especially in the context of our virtual environment experiments. As the link bitrate decreases to 1 Mbps and values below it, we encounter a unique challenge. In the virtual environment, where the processing delay for each packet can vary significantly, estimating the delay introduced solely by recirculation becomes a daunting task. The intrinsic variability in packet processing times makes it challenging to

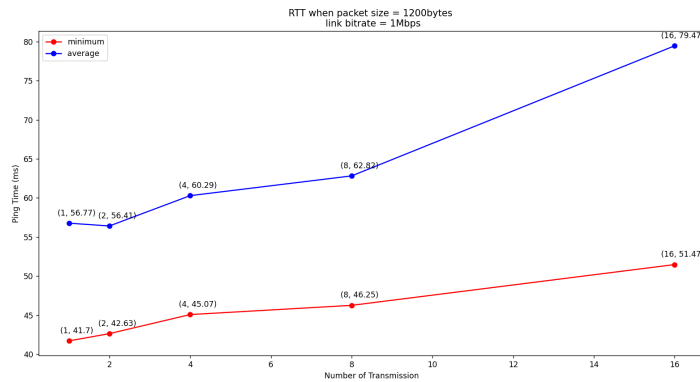


Figure 5.12. Minimum and average experimental RTT in the virtual Linux machine in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 1 Mbps

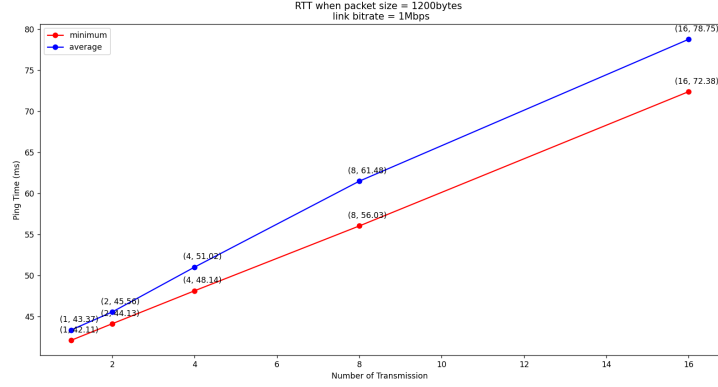


Figure 5.13. Minimum and average experimental RTT in the Linux Server in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 1 Mbps

precisely quantify the impact of recirculation when controlling packet delay, particularly in the virtual environment with low link bitrates.

In the graph showing RTT measurements at a 1 Mbps link bitrate in the virtual environment (see Figure 5.12), we observe an initial increase in packet delay from 41.7 to 42.63 ms with the introduction of the first recirculation, indicating a 0.96 ms difference. However, this pattern increases with subsequent recirculations. Over the next six recirculations, the delay grows from 42.93 to 46.25 ms, marking a 3.32 ms difference. Dividing this difference across the six recirculations reveals an additional delay of approximately 0.55 ms per recirculation. This variance aligns with our earlier observations, illustrating

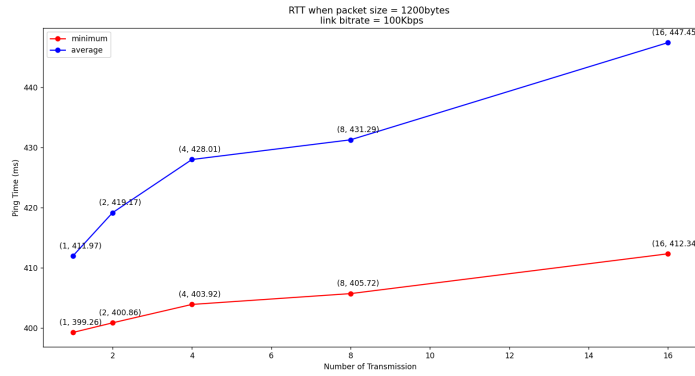


Figure 5.14. Minimum and average experimental RTT in the virtual Linux machine in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 100 kbps



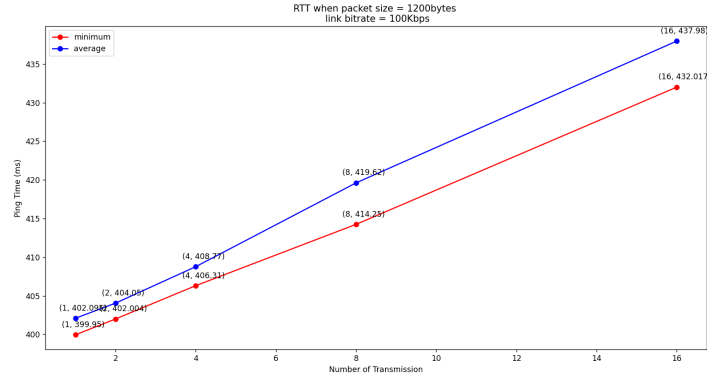


Figure 5.15. Minimum and average experimental RTT in the Linux Server in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 100 kbps

that different packets experience varied processing delays within the switch.

A similar trend is observed in the graphs representing RTT measurements at link bitrates of 100 Kbps and 10 Kbps in the virtual environment. At 100 Kbps (see figure 5.14), the first recirculation elevates packet delay from 399.26 to 400.86 ms, resulting in a 1.6 ms difference. Subsequent recirculations lead to a delay increase from 400.86 to 405.72 ms, marking a 4.86 ms difference. The additional delay introduced by each recirculation approximates 0.81 ms when evenly distributed across the six recirculations.

At a link bitrate of 10 Kbps (see figure 5.16), the initial recirculation raises packet

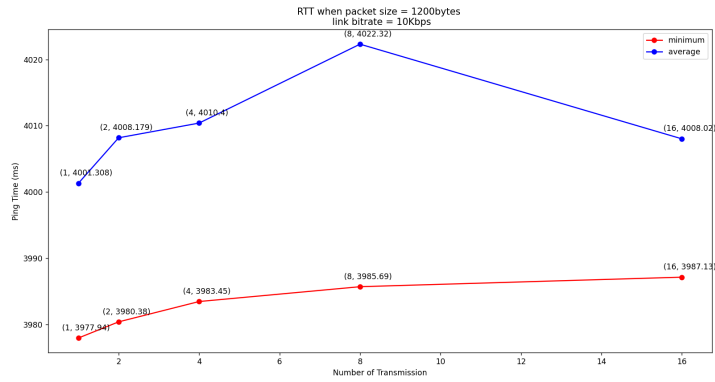


Figure 5.16. Minimum and average experimental RTT in the virtual Linux machine in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 10 kbps

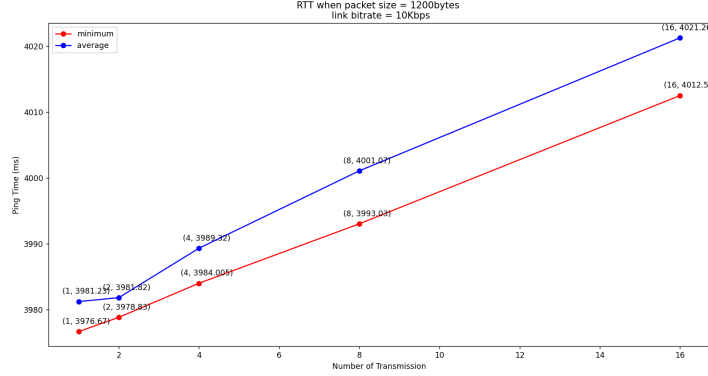


Figure 5.17. Minimum and average experimental RTT in the Linux Server in Ping experiments when the packet size is 1200 bytes for the different number of recirculations in the BMv2 switch with changing the link bitrates to 10 kbps

delay from 3977.94 to 3980.38 ms, reflecting a 2.44 ms difference. However, this pattern increases with subsequent recirculations, with the delay increasing from 3980.38 to 3985.69 ms over the next six recirculations, resulting in a 5.31 ms difference. Distributing this difference across the six recirculations shows that each recirculation adds less than 1 ms to the delay. The experiments conducted on the Linux server, on the other hand, exhibit a more predictable behavior when link bitrates decrease. This predictability is a consequence of the advantages offered by a real environment where we can estimate and anticipate processing and transmission delays more effectively, even in scenarios with low link bitrates. The graph in this context (see figure 5.13) reveals that the processing delay introduced by recirculation is approximately 2.02 ms, a value closely aligned with the processing delay observed in experiments conducted with a 1 Gbps link bitrate (figure 5.11) and those without specifying link bitrates (figure 5.9) in the Linux server environment, which was approximately 2.11 ms.

Furthermore, it's notable that the observed linear increase in delay with an increasing number of recirculations persists when the link bitrate drops to 100 kbps (see figure 5.15) and 10 kbps (figure 5.17). In these experiments, the impact of each recirculation remains consistent, adding approximately 2 ms of delay. These findings on the Linux server underscore the precision with which we can control packet delay on a real Linux server, even when link bitrates are extremely low. This outcome carries significant implications for our research, as it confirms our ability to manipulate packet delay in a controlled and predictable manner within a real-world network infrastructure.

### 5.2.3 Exploring the Trade-offs: Delay Control and Network Throughput

This section embarks on a comprehensive investigation into the delicate balance between delay control through packet recirculation and its corresponding implications on network

throughput—a crucial facet of network performance. Network throughput, an important performance parameter, is the volume of data packets successfully transmitted through the network per unit of time. Our aim here is to go into how the pursuit of efficient delay control influences the network's throughput and explore the resulting trade-offs.

We investigate the influence of packet recirculation on network throughput using the IPERF tool, a robust network performance measurement utility. This investigation explores the changes in network performance under varied recirculation instances within the BMv2 switch using a series of designed tests.

The methodology used in these experiments is similar to that used in earlier sections. To simulate various network situations, we change the number of packet recirculations in the BMv2 switch to 0, 1, 3, 7, and 15 while adjusting link bitrates. We provide the findings of our experiments using both TCP and UDP protocols.

In TCP mode, IPERF utilizes the Transmission Control Protocol to simulate real-world communication dynamics. This mode evaluates the maximum achievable throughput for a given connection by simulating data streams from the client to the server. This simulation aligns with the workings of TCP itself, where the protocol dynamically determines the sending rate.

UDP mode, on the other hand, examines available bandwidth and packet loss rates by sending UDP packets from the client to the server. This protocol skips TCP's error-checking and congestion control in order to improve network latency. To appropriately analyze the network's behavior in this situation, it is critical to specifically describe the sending rate.

Notably, in these experiments, data collection is performed across 30 IPERF experiments for each scenario, with the results being averaged for accuracy. The IPERF experiment configuration involves designating one host as the server and another as the client. It's essential to highlight that the reported results are representative of the client-side perspective when it receives the packets from the server. You can see the script for running IPERF experiments with a TCP connection in [Figure 5.18](#).

The experimental structure maintains the ordering of recirculation instances—0, 1, 3, 7, and 15—as established in previous sections. The range of link bitrates explored in the IPERF experiments changes as follows: for TCP, the link bitrates encompass "unspecified," 25 Mbps, and 10 Mbps; for UDP, the rates consist of "unspecified" and 10 Mbps. The chosen command for server-side execution in TCP mode is `iperf -s (-u)`, where the `-u` flag denotes the utilization of the UDP protocol. On the client side, the command follows the pattern `iperf -c (server-ip) -t 10 (-b X) -r (-u)`—where `-t` specifies the test duration, `-b` sets the sending rate, and `-r` measures performance in both upstream and downstream directions. The latter option specifically assesses the recirculation impact on throughput concerning both client-to-server and server-to-client transmissions.

First, we investigate the results when the protocol that we use for the Iperf connection is UDP. For UDP, one time we did not set the link bitrates. You can see the graphs for different sending rates in [Figure 5.19](#)

As you can see, when we did not specify any link bitrate, the maximum throughput that we could achieve when there is no packet recirculation in the programmable switch

```
#!/bin/bash

# Array to store individual bandwidth values
sbandwidths=()
rbandwidths=()

echo "iperf experiment when using TCP"

for i in {1..30} # Run the experiment 30 times
do
    # Run iperf and capture the output
    result=$(iperf -c 10.0.1.1 -t 10 -r )

    # Extract the bandwidth value from the iperf output
    sbandwidth=$(echo "$result" | awk '/sec/{print $(NF-1), $(NF)}'|tr -d '[:alpha:]\/'| head -n 1 )
    rbandwidth=$(echo "$result" | awk '/sec/{print $(7)}'|tr -d '[:alpha:]\/'| tail -n 1 )
    echo "Sender Bandwidth: $sbandwidth"
    echo "Receiver Bandwidth: $rbandwidth"

    # Add the bandwidth to the array
    sbandwidths+=("$sbandwidth")
    rbandwidths+=("$rbandwidth")
done

# Calculate statistics for receiver bandwidth
min_rbandwidth=$(printf '%s\n' "${rbandwidths[@]}" | sort -n | head -n 1)
max_rbandwidth=$(printf '%s\n' "${rbandwidths[@]}" | sort -n | tail -n 1)

echo "Receiver Bandwidth Statistics:"
echo "Minimum Receiver Bandwidth: $min_rbandwidth"
echo "Maximum Receiver Bandwidth: $max_rbandwidth"
for bandwidth in "${rbandwidths[@]}"
do
    total_rbandwidth=$(awk "BEGIN {print $total_rbandwidth + $bandwidth}")
done
average_rbandwidth=$(awk "BEGIN {print $total_rbandwidth / ${#rbandwidths[@]}}")

echo "Average Receiver Bandwidth: $average_rbandwidth"

sum_rbandwidth_sq=0
for bandwidth in "${rbandwidths[@]}"
do
    diff=$(awk "BEGIN {print $bandwidth - $average_rbandwidth}")
    sum_rbandwidth_sq=$(awk "BEGIN {print $sum_rbandwidth_sq + ($diff)^2}")
done
variance_rbandwidth=$(awk "BEGIN {print $sum_rbandwidth_sq / (${#rbandwidths[@]} - 1)}")
stddev_rbandwidth=$(awk "BEGIN {print sqrt($variance_rbandwidth)}")
echo "Standard Deviation for Receiver Bandwidth: $stddev_rbandwidth"
```

Figure 5.18. IPERF using the TCP connection experiment’s script

is close to 10 Mbps. So for the next experiment, I set the link bitrate to 10 Mbps to see the variations in throughput more precisely, so we can analyze them better. You can see the results for this chain of experiments in Figure 5.20

**Analyze:** In our IPERF experiments with UDP connections without specifying a link bitrate, several interesting trends emerge. When the sending rate of the source host is set to 10 Mbps, we notice that the most achievable throughput is approximately 9.94 Mbps. This is slightly less than the source’s sending rate due to the additional headers added by UDP, which reduce the effective throughput to about 96% of the available bandwidth.

As we increase the number of recirculations, particularly in scenarios where both the sending rate and link bandwidth are 10 Mbps, we observe a decrease in throughput. This phenomenon occurs for two main reasons:

*Delay-Induced Challenges:* With recirculation enabled in the BMv2 switch, a delay is introduced. For instance, if we initially send 10 Mbps of data in the first 10 seconds, the destination does not receive the entire 10 Mbps due to the delay caused by recirculation. This leads to an underutilization of the available bandwidth.

*Link Saturation:* Additionally, the load on the link connected to the output port of

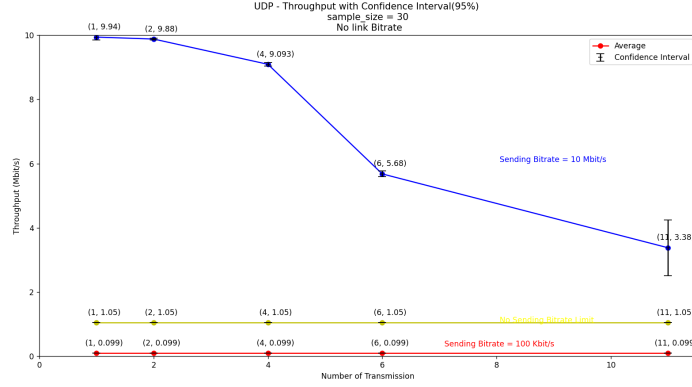


Figure 5.19. Average experimental throughput in IPERF experiments when using UDP connection with different sending rates for the different number of recirculations in the BMv2 switch without specifying any link bitrate

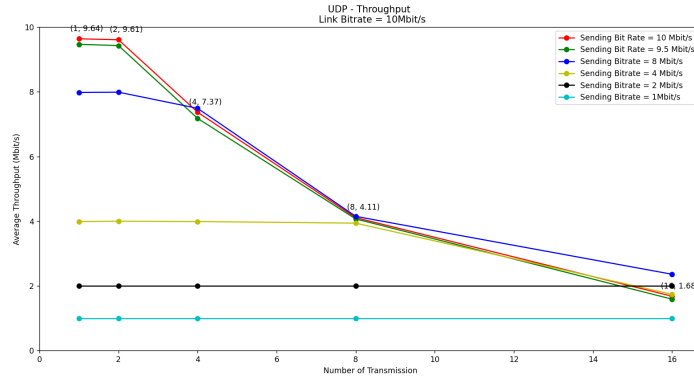


Figure 5.20. Average experimental throughput in IPERF experiments when using UDP connection with different sending rates for the different number of recirculations in the BMv2 switch with the link bitrate equal to 10 Mbps

the switch periodically exceeds the link's capacity, especially when recirculation occurs. This overload results in packet loss, further reducing the achieved throughput.

However, this decrease in throughput is not universally applicable. For different sending bitrates, we notice varying behavior:

When we decreased the link bitrate to 10 Mbps and maintained a sending rate of 10 Mbps and 9.5 Mbps, we observed a throughput reduction with just one recirculation due to the overload on the output link.

However, at a sending rate of 8 Mbps, even with one recirculation, we did not experience packet loss, allowing the source host to fully utilize its capacity before the throughput reduction occurred.

A similar situation occurred at a sending rate of 4 Mbps, but the throughput reduction only became evident after seven recirculations.

For sending rates of 2 Mbps and 1 Mbps, the throughput remained consistent until 15 recirculations. Based on this trend, we anticipate a throughput reduction at a sending rate of 2 Mbps when recirculation increases to 16.

After analyzing the results for UDP, it is time to see the results for IPERF when it uses the TCP connection. In TCP, we do not change the sending rate because the TCP protocol sets this rate automatically based on the mechanism that it uses to provide a reliable connection. In this series of experiments, we changed the link bitrates, and first, we did not set any link bitrate. Then, based on the results that I got for the state that the number of recirculations was 0 and I got something about 25 Mbps for throughput in the network, I repeated the experiments with setting the link bitrate to 25 Mbps. and at last, I set it to 10 Mbps in order to have a comparison with the results that we have done in UDP. You can see the comparison between these results in Figure 5.21

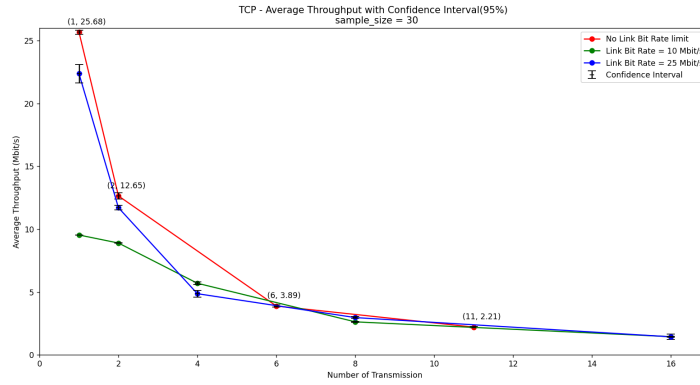


Figure 5.21. Average experimental throughput in the IPERF experiments when using a TCP connection for the different number of recirculations in the BMv2 switch with different link bitrates

In our IPERF experiments with TCP connections, we maintain a constant sending rate. TCP inherently adjusts its sending rate based on congestion control mechanisms and other factors. Therefore, our focus here is on observing how different link bitrates and the number of recirculations impact network throughput.

In this scenario, TCP aims to utilize the available bandwidth to achieve the highest throughput possible. When recirculation occurs and the data load on the output port of the switch increases, TCP's congestion control mechanisms respond by reducing the sending rate to prevent congestion and packet loss. Consequently, we observe that as the number of recirculations increases, the throughput decreases continuously.

#### 5.2.4 Enhanced Complexity: Dual Switch Recirculation

Transitioning into an expanded network topology, this section deals with the complexities introduced by incorporating an additional BMv2 switch. With the evolved structure,

packets now traverse a two-switch path to their destination, a configuration that demands an exploration of delay control possibilities and their impact on both delay and network throughput. By introducing the dual-switch scenario and deploying recirculation within each BMv2 switch, we aim to uncover the intricacies of managing packet delay across this new topology.

An important aspect of this investigation is the comparison between single-switch and dual-switch networks. This strategic contrast provides an atmosphere for understanding the impact of recirculation within a more complicated network design. We want to know how the cumulative impacts of dual-switch packet recirculation differ from the simpler single-switch version.

Incorporating two BMv2 switches into the network introduces a level of complexity that prompts questions about the interplay between these switches and their combined impact on packet delay and network throughput. This section bridges that knowledge gap, offering insights into delay control within a network involving multiple switches.

Using experimental findings, we examine how packet delay is controlled in a dual-switch scenario. This investigation looks at the impact of recirculation on both delay and network throughput.

Now let's investigate the results of these experiments:

The first graph 5.22 shows the average delay of packets when using Ping experiments 1000 times with two BMv2 switches and without specifying any link bitrate and the comparison with the results with a single switch.

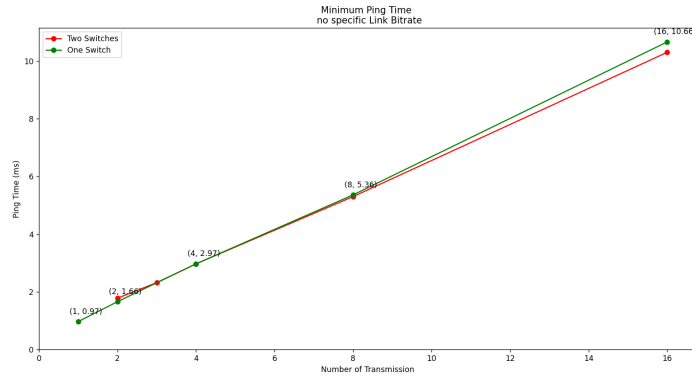


Figure 5.22. The minimum delay of packets for the network with two BMv2 switches vs. the delay of packets for the network with a single switch without specifying any link bitrate

In this set of results, we explore the delay introduced by two consecutive programmable switches compared to a single switch within the virtual environment. The x-axis represents the number of transmissions, which is one more than the number of recirculations in the scenario that the network topology contains one programmable switch. In the case of two switches, the number of transmissions is the sum of recirculations in each switch plus two. For instance, when there are two transmissions, it corresponds to one recirculation for a single-switch scenario and zero recirculation for each of the two switches in

the dual-switch scenario.

In Figure 5.22, which illustrates the comparison between minimum RTT results in Ping experiments with varying numbers of transmissions when no link bitrate is specified, the lines closely parallel each other. This alignment indicates that under these conditions, the delay of one recirculation introduced by one switch in the network with one programmable switch scenario is almost equivalent to the processing delay of one switch within a network having two programmable switches. For example, with one recirculation in the BMv2 switch in the single switch scenario, the minimum delay in the network is nearly equal to the delay when there's no recirculation in the switches in the scenario with two switches in the network. This value is approximately 1.66 ms, and this pattern persists as the number of recirculations increases.

Another important finding from these results is that in scenarios with no recirculation and the packets just traversing the switches with their inherent processing delays, the delay in the single switch network scenario is 0.97 ms. In contrast, in the dual-switch network scenario, this delay is 1.66 ms. This 0.69 ms difference highlights the additional processing time introduced by an extra switch for packet forwarding within a programmable switch. Essentially, this difference signifies the extra processing time incurred when a packet traverses one more programmable switch. As discussed earlier, the starting point of the delay graph for scenarios with no recirculation in a single switch network, without specifying any link bitrate, encompasses the total processing time of all components, including hosts and the BMv2 switch, which is 0.97 ms.

In conclusion, subtracting the 0.69 ms processing time of the BMv2 switch from the 0.97 ms delay for the scenario with a single switch reveals that the remaining 0.28 ms accounts for the processing time of the source and destination hosts.

The second graph 5.23 shows the minimum delay of packets when using Ping experiments 1000 times with two BMv2 switches and with the link bitrate equal to 10 Mbps and the comparison with the results with a single switch. I did not put the results when I changed the link bitrate to other variables because they did not give us any extra conclusion.

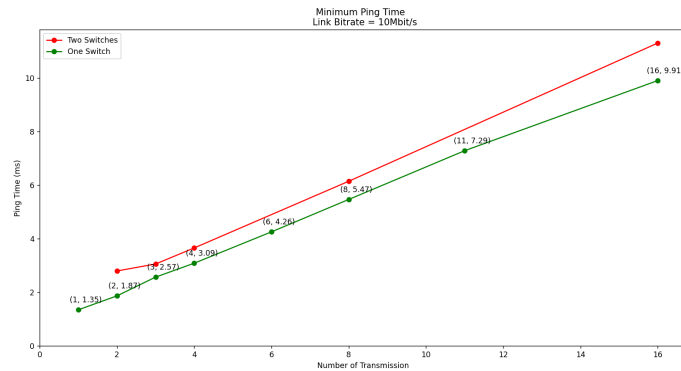


Figure 5.23. The minimum delay of packets for the network with two BMv2 switches vs. the delay of packets for the network with a single switch with a link bitrate of 10 Mbps



The results presented in Figure 5.23, which compares delays in scenarios with one switch and two switches while varying the link bitrate to 10 Mbps, align closely with our previous findings. However, it's important to note a slight difference observed this time, where the delay for packets traversing two switches is marginally higher. This discrepancy can be attributed to the phenomenon I previously explained. When we reduce the link bitrates, the processing time of network components increases. In the scenario with two switches, we have an additional switch, which results in a greater number of components. Consequently, with the link bitrates decreasing, the processing time increases more noticeably due to the larger number of components in the dual-switch setup.

The results for measuring average throughput using IPERF with the TCP connection with different link bitrates are shown in Figure 5.24, and the comparison for the throughput when we use IPERF with TCP between the topology with a single switch and with two switches can be seen in these different states: without specifying any link bitrate 5.25, link bitrate = 25 Mbps 5.26 and link bitrate = 10 Mbps 5.27

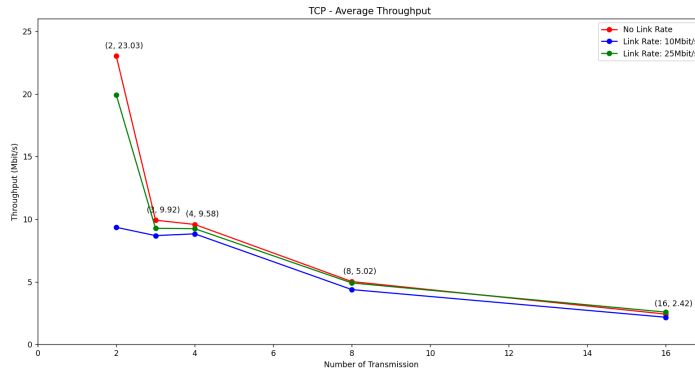


Figure 5.24. This figure displays the results of measuring average throughput using IPERF with a TCP connection under different link bitrates in the scenario that the network consists of two programmable switches.

In Figure 5.24, the graphs illustrate a consistent decrease in throughput as the number of recirculations increases. In all the experiments, the source host wants to send data at the highest capacity allowed by the network connections, conforming to the techniques used by the TCP connection for congestion control and packet loss prevention. When recirculation happens within the switch, however, the system's load increases, causing TCP to restrict the data transfer rate. This behavior is similar to instances in which the network has a single programmable switch.

It is interesting to note that the maximum achievable throughput of the network begins at a different point in each scenario of link bitrate setting. For scenarios without link bitrate specification, this starting throughput is 23.03 Mbps. For a link bitrate of 25 Mbps, the maximum throughput decreases to 21.7 Mbps, and for a link bitrate of 10 Mbps, it drops to 9.39 Mbps. However, as the number of recirculations increases,

these maximum achievable throughput lines begin to converge, reflecting a reduction in throughput across different link bitrates."

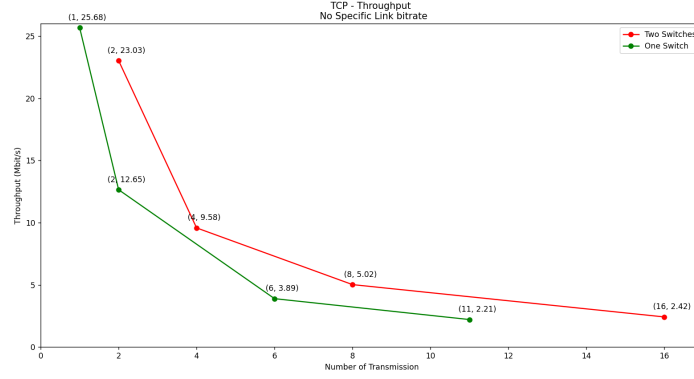


Figure 5.25. The figure illustrates the throughput comparison between network topologies with a single BMv2 switch and two BMv2 switches when no specific link bitrate is specified. The connections are TCP.

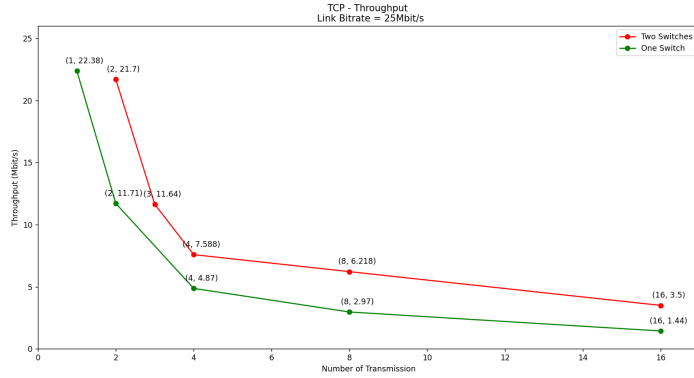


Figure 5.26. The figure illustrates the throughput comparison between network topologies with a single BMv2 switch and two BMv2 switches when the link bitrate is 25 Mbps. The connections are TCP.

In the following comparisons of throughput measured through IPERF experiments, where one scenario involves a single switch in the network and the other entails two switches, a consistent trend emerges. For all link bitrates, both graphs show a similar decrease in throughput as the number of transmissions increases. However, an important distinction lies in the achievable throughput at each point on these graphs.

In scenarios with two switches, the throughput is consistently higher than in scenarios with only one switch. This difference arises due to the additional recirculation introduced by the presence of one switch in the network. With the same number of transmissions,

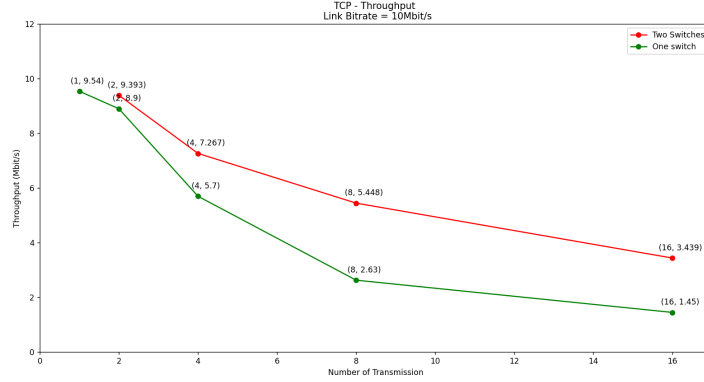


Figure 5.27. The figure illustrates the throughput comparison between network topologies with a single BMv2 switch and two BMv2 switches when the link bitrate is 10 Mbps. The connections are TCP.

the single-switch scenario involves one more recirculation. Consequently, this leads to an increased load on the output port of the switch and prompts a more aggressive TCP rate reduction to avoid congestion and packet loss.

In summary, while both scenarios exhibit a decreasing throughput trend as the number of transmissions increases, the scenario with one switch consistently achieves lower throughput at each data point due to the additional recirculation and associated congestion control measures.



## Chapter 6

# Conclusion

The primary goal of this thesis was to leverage the capabilities of programmable switches for the precise control of packet delays within a network. The thesis attempted to answer a basic question: how can we modify packet order in network service chaining by using data plane programmability? Drawing on a foundation of knowledge concerning programmable switches and their data plane programmability, we began the path of understanding their inner workings and potential.

Extensive research and learning were required to comprehend the behavior of programmable switches and the complexities of the P4 language. We set out on an endeavor to get a thorough understanding of these technologies in order to maximize their potential for our research goals. This first research part was critical since it set the framework for the development of novel solutions.

Our exploration led to the realization that programmable data planes could offer a solution. The concept of postponing packet transmission within a programmable switch until the arrival of another packet, thus allowing us to reorder packets at the destination, took shape.

The key to this reordering strategy was packet recirculation, which introduced controlled delays in packet transmission. We embarked on a quest to develop and implement this recirculation concept within programmable switches. Two implementations were considered: the first, a more general approach, involved recirculating all received packets, allowing us to investigate the impact of multiple recirculation cycles on packet delay. The second, a more specific approach, focused on recirculating custom packets while forwarding others normally, accompanied by additional headers for custom packets.

Following the implementation of packet recirculation, the next phase involved evaluating its outcomes. To achieve this, we employed Mininet to simulate diverse network topologies. The initial segment of the evaluation focused on determining the baseline for packet delay. In this part, a simplified network topology was employed, consisting of a single BMv2 programmable switch situated between a source host and a destination host. Crucially, no recirculation was applied inside the programmable switch. This approach allowed us to investigate the intrinsic processing delay introduced by the network components, particularly the programmable switch and the influence of transmission delay on packet delays for various link bitrates.

The second segment was the most crucial, as it goes deeply into the heart of the research question: how to effectively introduce controlled delays in packet transmission. Here, the goal was to assess the impact of recirculation within the programmable switch on packet delays. The aim was to identify how many recirculations the programmable switch should apply to packets under different network configurations, including different link bitrates, to achieve the desired delays. I assessed the effect of packet recirculation on packet delay using the Ping tool. Furthermore, we examined how packet recirculation influenced network throughput by employing the Iperf tool for both TCP and UDP connections.

This evaluation was performed in both virtual Linux environments and on a real Linux server. The aim was to compare and contrast the results obtained in these two distinct environments and demonstrate the adaptability of the mechanism implemented in this research. By showcasing results in both virtual and real environments, this work serves as a valuable resource for future investigations and research endeavors, providing understandings that bridge the gap between the virtual and real networking worlds.

The study expanded to incorporate multiple programmable switches within the topology, allowing us to explore the effects of employing multiple programmable switches on packet delay and network throughput. In summary, this study investigates how programmable switches can be used to apply exact control over packet delays in network environments. It opens the door to further research and innovation in the constantly evolving domain of software-defined networking, offering the potential to reshape network and service delivery for a wide range of applications.

# Bibliography

- [1] Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, 54(4), may 2021.
- [2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.
- [3] Bhargavi Goswami, Manasa Kulkarni, and Joy Paulose. A survey on p4 challenges in software defined networks: P4 programming. *IEEE Access*, 11:54373–54387, 2023.
- [4] Ya Gao, Zhenling Wang, and Sang-Bing Tsai. A review of p4 programmable data planes for network security. *Mob. Inf. Syst.*, 2021, jan 2021.
- [5] Francesco Paolucci, Davide Scano, Piero Castoldi, and Emiliano De Paoli. Latency control in service chaining using p4-based data plane programmability. *Computer Networks*, 216:109227, 2022.
- [6] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free network ordering for strongly consistent distributed applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 293–320, Boston, MA, April 2023. USENIX Association.
- [7] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '20*, page 274–281, New York, NY, USA, 2020. Association for Computing Machinery.
- [8] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, number 12, pages 313–323, 2018.
- [9] Belma Turkovic, Fernando Kuipers, Niels van Adrichem, and Koen Langendoen. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, NEAT '18*, page 45–51, New York, NY, USA, 2018. Association for Computing Machinery.
- [10] Ralf Kundel, Jeremias Blendin, Tobias Viernickel, Boris Koldehofe, and Ralf Steinmetz. P4-codel: Active queue management in programmable data planes. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–4, 2018.