UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

NETWORK CODING DATA PLANES WITH PROGRAMMABLE SWITCHES

Diogo Figueiredo Pinto

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Dr. Fernando Manuel Valente Ramos
e co-orientada pela Prof.ª Dra. Muriel Médard

2017

Acknowledgments

I would like to begin with a special thanks to my advisors Professor Fernando Manuel Valente Ramos and Professor Muriel Médard, for all the patience, support and confidence throughout the last year.

A flimsy one to all my colleagues, for choosing not to use the 1.3.19 office during all the year, which allowed me to have all the peace, silence and concentration, in the world.

I am also thankful to the faculty's facilities, for providing me with microwaves to heat my delicious homemade meals.

Finally, a special one to Marlboro, for the marvelous cigarettes that helped me keep my mental sanity.

FUNDING This work was partially supported by the European Commission through project FP7 SEGRID (607109) and project H2020 SUPERCLOUD (643964), and by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/CEC/00408/2013 (LaSIGE).

Resumo

Atualmente, as redes de computadores seguem um paradigma tradicional de *store-and-forward*, ou seja, os dispositivos de rede fazem armazenamento, encaminhamento e/ou replicação de pacotes recebidos, sem os modificar. No virar do milénio, surgiu um artigo seminal [24], no qual foi demonstrado teoricamente que a *combinação* da informação proveniente de diversos pacotes, permite aumentar a capacidade de uma rede relativamente à capacidade máxima, alcançada por simples encaminhamento. Este resultado representou o nascimento de uma área promissora de investigação, conhecida como Codificação na Rede *(Network Coding)*. A ideia é permitir que os nós intermédios da rede, possam aplicar uma função de codificação sobre o conteúdo dos pacotes antes do seu encaminhamento, proporcionando assim um novo paradigma de *store-code-forward*.

A família de técnicas tradicionais pode ser divida em duas categorias, com propósitos distintos. Codificação na Origem *(Source Coding)* com o objetivo de comprimir a informação enviada, e Codificação no Canal *(Channel Coding)* para compensar perdas e alteração de informação em canais ruidosos. Com codificação na rede, surge oportunidade para a definição de técnicas mais elaboradas e que visam outros propósitos. Deste modo, as técnicas de codificação tradicionais podem ser extendidas para além da codificação de pacotes em nós de origem, e da descodificação em nós de destino.

De um ponto de vista geral, a codificação na rede tem potencial para melhorar a taxa de transferência de informação na rede; aumentar a resiliência contra perda de pacotes, interrupção de canais e nós da rede; e aumentar a segurança contra ataques maliciosos que visam a captura, interpretação e modificação de pacotes.

Como técnica, a codificação na rede pode ser aplicada de dois modos distintos. Por um lado, sobre pacotes provenientes de um único fluxo de comunicação *(intraflow network coding)* e por outro, sobre múltiplos fluxos sem qualquer relação entre si *(interflow network coding)*.

A título de exemplo, se considerarmos dois fluxos que chegam a um *switch* por dois canais distintos, mas que contestam o mesmo canal de saída, temos um gargalo na rede. Usando codificação na rede, o *switch* pode aplicar, bit a bit, o Ou-Exclusivo (XOR) sobre dois pacotes (um de cada fluxo) e encaminhar o resultado. A taxa de transferência é

melhorada, pois o *switch* necessita apenas de encaminhar um pacote codificado em vez de dois originais. É de salientar que, de forma a descodificar o pacote, o nó de destino tem de ter um dos pacotes originais usados na codificação.

Portanto, as vantagens da codificação na rede estão dependentes da topologia da rede, da própria função de codificação utilizada, e do modo como é aplicada.

Numa rede, um nó intermédio terá à partida acesso a vários pacotes. De forma a tirar máximo partido da técnica de codificação na rede, as funções de codificação utilizadas acabam por consistir num código linear *(Linear Network Coding)*. A ideia é considerar todos os pacotes de uma mensagem a enviar (por exemplo, um ficheiro de texto, um vídeo, ou até um simples pedido HTTP) como um vetor de elementos de um dado campo finito. O tamanho de cada elemento, é dado pelo número de bits necessário para representar o maior valor desse campo. Se por exemplo o campo finito for 256, cada elemento terá 8 bits. A um vetor de elementos, damos o nome de símbolo.

Associado a cada símbolo transmitido na rede, existe um vetor de coeficientes, necessário para codificação e descodificação. O tamanho do vetor, é ditado pelo número de símbolos originais. Se a mensagem é divida em 5 símbolos, então o vetor tem tamanho 5.

Para codificar e criar um novo símbolo, o nó da rede começa por selecionar um novo vetor de coeficientes local. A função de codificação consiste numa combinação linear sobre um dado número de símbolos, utilizando o novo vetor local. O vetor do novo símbolo codificado é obtido da mesma forma. Sobre os vetores dos símbolos utilizados, é feita uma combinação linear utilizando o vetor local.

Para descodificar os símbolos originais, são necessários um número igual de símbolos codificados, linearmente independentes. De forma a que os símbolos codificados e recodificados na rede, sejam linearmente independentes, podem ser utilizados algoritmos de tempo polinomial [59], para estabelecer os vetores locais utilizados por cada nó intermédio da rede. De forma a simplificar o problema, os vetores locais podem ser aleatórios *(Random Linear Network Coding)*. Se o campo finito for suficientemente grande, a probabilidade de obter símbolos codificados linearmente independentes chega perto dos 100%.

De forma a ter vetores mais reduzidos, tornando as operações mais simples, e permitindo uma descodificação gradual, os símbolos originais da mensagem podem ser organizados em gerações. Por cada geração, são gerados e injetados pela rede, símbolos codificados. Quando uma geração é descodificada, procede-se para a geração seguinte.

Repare-se que a função de codificação referida anteriormente, com base em XOR, é o caso base e mais simples de um código linear. Neste caso, o campo finito é de tamanho 2.

Apesar de ser um conceito relativamente simples, implementar e usar técnicas de codificação no plano de dados dos próprios dispositivos de rede é uma tarefa bastante complicada.

Até mesmo quase impossível na maioria dos casos, visto que a *payload* dos pacotes é sujeita a alterações. O seu funcionamento baseia-se em protocolos fixos, que correm no próprio hardware de forma a maximizar o desempenho, o que torna difícil a tarefa de configurar e gerir uma rede para além das simples operações de encaminhamento de pacotes. Por este motivo, as implementações práticas de codificação na rede que têm vindo a surgir nos últimos anos, operam em redes *overlay*. Uma rede *overlay* reside logicamente na camada de aplicação, implicando que os dispositivos de rede propriamente ditos não são alterados.

O interesse crescente em operações mais complexas e exigentes na rede, mas condicionado pelo funcionamento rígido e fechado dos *routers* e *switches* tradicionais, motivou uma mudança de paradigma: de redes configuráveis para redes programáveis. A primeira instância de uma rede programável é conhecida como Rede Definida por Software *(SDN)*. Numa rede SDN, o plano de controlo é separado do plano de dados, e reside num dispositivo à parte - um controlador logicamente centralizado. Utilizando a informação de pacotes provenientes do plano de dados dos *switches*, o controlador pode definir políticas de configuração mais flexíveis e instalar regras nas tabelas *match-action* dos mesmos. A comunicação entre os *switches* e o controlador está estandardizada, sendo utilizado um protocolo conhecido como *OpenFlow*. A limitação de *switches* e controladores *OpenFlow* está no processamento de pacotes, que continua a ser fixo. De facto, o *OpenFlow* atua sobre um conjunto fixo de protocolos. Além disso, a sequência de tabelas e ações de um *switch Openflow* também é fixa. Portanto, o *OpenFlow* não permite realmente definir nova funcionalidade no plano de dados de um *switch*. Apenas fornece um meio para o controlador tomar decisões e instalar regras nas tabelas *match-action*, dos mesmos. No âmbito de codificação na rede, este fator impossibilita a alteração da *payload* dos pacotes, e consequentemente a sua combinação.

No entanto, têm vindo a surgir recentemente *switches* programáveis, alguns até já em produção (por exemplo, Tofino da Barefoot Networks). Estes dispositivos permitem a programação e reprogramação do plano de dados, o que possibilita uma definição precisa e customizada do modo de processamento de pacotes. Com esta liberdade, a codificação na rede torna-se possível, no plano de dados.

Porém, a sua programação é baseada em interfaces de baixo nível, tornando-se um processo demorado e doloroso. Esta dificuldade, acrescida também às limitações descritas do *OpenFlow*, motivou a criação da linguagem de alto nível, P4.

A linguagem P4 permite definir cabeçalhos, *parsers* e a sequência de tabelas de *match-action*, para qualquer dispositivo de rede compatível. As ações podem ser definidas utilizando um conjunto de primitivas básicas oferecidas pela linguagem. A linguagem P4 oferece três vantagens. Primeiro, não está dependente de protocolos e formatos de pacotes específicos, uma vez que a sua definição pode ser feita pelo programador. Segundo, per-

mite a reconfiguração do *switch* a qualquer momento. Terceiro, não depende do hardware subjacente, podendo ser escrita, da mesma forma, para qualquer dispositivo que tenha o compilador adequado.

O objetivo desta dissertação consiste no desenho, implementação e avaliação do primeiro *switch* capaz de realizar codificação no plano de dados, recorrendo à linguagem P4. Mais concretamente, a nossa solução consiste em dois *switches*: um que executa XOR (P4-XOR Switch), e outro que executa uma variante de *Random Linear Network Coding* (P4-RLNC Switch). Durante a implementação enfrentámos vários desafios, devido às peculiaridades da linguagem. Entre os principais fatores que dificultaram a implementação, está o facto de a linguagem ser declarativa, não permitindo a criação de estruturas de dados auxiliares em tempo de execução; e a impossibilidade de criar ciclos, essencial para repetir o mesmo processo de codificação sobre os vários elementos dos símbolos, no caso do P4-RLNC Switch.

Sendo um trabalho inovador, a avaliação focou-se essencialmente na funcionalidade dos dois *switches* concretizados. Adicionalmente, a performance do P4-XOR Switch também foi avaliada.

Palavras-chave: Redes definidas por Software, OpenFlow, Codificação na Rede,
Comutadores Programáveis, P4

Abstract

Network Coding (NC) is a technique that can be used to improve a network's throughput. In addition, it has significant potential to improve the security, manageability, resilience (to packet losses, link failures and node departures) and the support of quality of service, in both wired and wireless network environments. The idea is to allow intermediate nodes of the network (i.e. switches and/or routers) to mix the contents of incoming data packets before forwarding them. Something that, traditionally carried out at source nodes, is therefore extended to the network, creating an array of new options.

The difficulty of deploying NC on traditional switches lies in the impossibility to change or extend their operation with the requirements of this new paradigm. The devices are closed, the software and underlying hardware are vendor specific, and follow a fixed set of protocols and processing pipeline. This rigidity precludes NC in today's switches and routers.

Fortunately, programmable switches are beginning to emerge, with some already achieving production-levels and reaching the market (e.g., Barefoot Tofino). A new high-level language to program these switches has recently been proposed: P4. The P4 language allows the precise definition of how packets are processed in these programmable switches. Namely, it enables the definition of headers, parsers, match-action tables, and the processing pipeline itself. Therefore, by taking advantage of these constructs, P4 enables the deployment of NC, on the switch's data plane, for the first time.

In this dissertation, we design and implement two NC switches using the P4 language. Both switches employ Linear Network Coding (LNC). The main difference is that the first (P4-XOR Switch), simply performs the XOR of packets (i.e., a linear code with field size 2). The second (P4-RLNC Switch) is more generic, allowing larger field sizes. For this purpose it performs Random Linear Network Coding (RLNC), which is a random variant of LNC.

The evaluation was performed on Mininet (a network emulator) and focused on the functionality of both switches. Additionally, the performance of the P4-XOR Switch was tested as well. The main conclusion is that our implementations correctly perform the required operations allowing, for the first time, NC to be performed in real data planes.

Keywords: Software-Defined Networking, OpenFlow, Network Coding, Programmable Switches, P4

Contents

List of Figures

List of Tables

Chapter 1 – Introduction

This chapter serves as an introduction for this dissertation. The motivation (§ 1.1), goals (§ 1.2) and contributions (§ 1.3) are discussed here. Lastly, the remaining of the document is outlined (§ 1.4).

In existing computer networks, packets are transmitted from source nodes to destination nodes through a chain of intermediate nodes, by a *store-and-forward* paradigm. In this paradigm, intermediate nodes simply store data packets from input links, and forward one or several copies to output links. Network Coding (NC) extends this paradigm to *store-code-forward*. Its premise is simple. Instead of simply store, replicate and/or forward incoming data packets, intermediate nodes are also allowed to mix the contents of several packets, by applying a coding function.

The encoding scheme can be designed in several different ways. It may be linear, non-linear, random, static, convolutional, etc. However, derived from the characteristics of computer networks, the encoding scheme tends to be: linear and fixed (Linear Network Coding [62]), or linear and random (Random Linear Network Coding [47]).

Depending on the encoding scheme and on the network topology, the NC technique may provide several advantages. The main advantage is an increase of throughput, over traditional routing, specially in multicast scenarios. It may also improve security (e.g., to packet sniffing), manageability, resilience (to losses of specific packets, link failures and node departures) and quality of service.

Due to its potential and significant advances on the theoretical front, several practical applications emerged over the years, in different areas. This includes, content distribution systems [68], wireless networks [51], ad-hoc networks [39], storage networks [36], peer-to-peer networks [61], disruption-tolerant networks [63], among others.

However, its deployment has proved to be problematic, in practice. This is mainly due to the difficulty to change, extend or adapt the functionality of traditional switches. The incorporated switching chips have a fixed processing pipeline, which operates over a fixed set of protocols. Therefore, until now, practical implementations of NC tend to operate as overlay networks. An overlay network is logically located at the application layer (i.e., running in end-hosts), meaning that the underlying physical network devices are not changed.

1.1 Motivation

As already mentioned, NC is very advantageous, in theory. However, enabling it in today's networks can be, in practice, a big challenge as current switches do not offer the required flexibility. The processing pipelines are unnecessarily long and contain tables fixed in number, arrangement and size, which process a fixed set of headers. This makes it hard to add new protocols or change the existing ones. Therefore, NC is not possible in hardware switches.

However, computer networks are changing paradigm: from configurable to programmable. This change finally brings the opportunity to deploy NC at the network layer.

Software-Defined Networks (SDN) [56] are considered as the first instance of a programmable network. The idea is to enable control plane programmability, by physically decoupling the data plane from the control plane. This decoupling provides a logically centralized view over the network, which permits a more flexible and on-the-fly allocation of resources, like buffer management and dynamic routing. The communication between the control plane and data plane is standardized, in a protocol known as OpenFlow [69, 14]. However, SDN only addresses the exposition of the data plane, not its programming. Likewise, OpenFlow-compliant switches still operate over a fixed set of protocols, and a fixed processing pipeline of match-action tables. Moreover, the set of primitive instructions that may be used to define an action, is also fixed. In addition, the set of protocols, primitive instructions and processing pipeline do not contemplate modifications to the packet's payload. Therefore, OpenFlow alone does not allow the encoding of packets required for NC.

Nevertheless, it is true that several works leveraged on the OpenFlow capabilities to deploy NC. However, all of them required extensions and modifications to the OpenFlow protocol itself. For instance, in [72, 85] OpenFlow is extended with additional actions; and in [82, 57] it is integrated with the KODO library [73].

Fortunately, there has been recently, a great development of programmable switches. With their low-level interfaces, its easier to customize the processing pipeline of match-action tables, within the data plane. This means that, they are capable of running new user created protocols. The ability to program switches, and the restrictions of previous solutions (e.g., OpenFlow), served as motivation for the creation of high-level languages. Their objective is to ease the task of switch programming, and focus on expressiveness and portability.

The first high-level language, for switch programming, is known as P4 [26]. The P4 language allows the definition of protocol headers, parsers, tables, actions and the processing pipeline itself. It is designed to be compiled to the underlying hardware. Many renowned companies are already starting to adopt P4 [17], and so it expected to become the standard for switch programming. In fact, P4 is already supported by the first programmable hardware switch: Barefoot Tofino [1].

Leveraging from the capabilities of programmable switches and languages such as P4, it is now possible to manipulate the payload of packets. Therefore, by using P4, we design and implement the first P4-compliant switch, capable of performing NC within the data plane itself.

1.2 Goals

The main goal of this dissertation is the design and implementation of a switch, capable of performing NC on the data plane. With that purpose, we use the recent and promising P4 language, which enables a custom and precise definition of the data plane of P4-compliant programmable switches. The target of our proposal is a P4-compliant software switch, provided by the P4 Consortium, known as Behavioral Model 2 (bmv2) [18]. This switch has been the target used in most related work, as in principle a program that runs in the bmv2 should run in any other target having the required compiler (e.g., Barefoot Tofino).

1.3 Contributions

The main contribution of this dissertation is the design and implementation of two NC data planes (P4-XOR Switch and P4-RLNC Switch), for P4-compliant switches, for the first time. As the names suggest, the P4-XOR Switch consists on Exclusive-OR Network Coding (XOR), and the P4-RLNC Switch on Random Linear Network Coding (RLNC). In order not to work just as a fixed code, but rather as a network protocol as well, we created and used a custom header stack, based on MPLS, to carry all encoding metadata. Besides the XOR and RLNC schemes, we also designed and implemented our own buffers, and forwarding rules. The implementation uses the P4-14 version of language, which is the most stable at this moment.

Finally, the evaluation of our solution was performed recurring to a network emulator with P4 support - Mininet. The evaluation includes functionality tests for both switches. Additionally, we also run performance tests, consisting of throughput and CPU utilization measurements, for the P4-XOR Switch, in the well-known butterfly topology. Our evaluation demonstrates both solutions to fulfill all requirements.

The result of this work is publicly available in our group's repository[1]. It includes the developed tools to automatically generate the P4 code of our switches; and the traffic generators, receptors and decoders, for testing purposes.

1.4 Structure of the document

This document is organized as follows. **Chapter 2** (§ 2) provides the background and related work on NC and P4. **Chapter 3** (§ 3) consists on the design of our NC solutions, using P4. **Chapter 4** (§ 4) details the implementation. **Chapter 5** (§ 5) presents the evaluation in terms of functionality and performance tests. **Chapter 6** (§ 6) finishes with a conclusion about the work done, and what may be done in the future.

[1] https://github.com/netx-ulx/NC.p4

Chapter 2 – Related Work

In this chapter, we overview the most important aspects, in the context of this dissertation, on Network Coding and the P4 language.

Network Coding (NC), as a technique, has a great potential to improve the overall performance of a network, over traditional routing. Specially, in a single-source multicast scenario. We start by specifying how the maximum multicast throughput is determined (§ 2.1.1) which, unlike with routing, may always be achievable with a NC solution. We introduce Exclusive-OR Network Coding, to exemplify this result (§ 2.1.2), and also because is one of the coding schemes we implemented. For the same reason, we describe Linear Network Coding (§ 2.1.3) and Random Linear Network Coding (§ 2.1.4). Then, we explore some of its variants (§ 2.1.5), advantages and use cases (§ 2.1.6). Finally, we finish with some practical applications of NC (§ 2.1.7).

With the advent of programmable switches and of the P4 language, NC can be deployed on the data plane of a switch. We contextualize the motivation for the emergence of the language (§ 2.2.1, § 2.2.2). Then briefly explain, in some detail, the language itself (§ 2.2.3). We close the section, with applications that use P4 (§ 2.2.4).

2.1 Network Coding - Overview

In today's existing packet computer networks such as the Internet, information is still delivered by routing packets from source to destination via intermediate nodes. It is the traditional store-and-forward design, i.e., intermediate nodes simply buffer received packets from input links and then relay or replicate them unmodified to output links. NC changes this design by allowing intermediate nodes to perform some computation, like applying a coding function or mix several packets before sending them to output links.

This mixing has been theoretically proven to maximize network throughput [24, 49, 54, 62]. For multicast, it can be done in a distributed manner with low complexity, and is robust to packet losses and network failures [46, 67]. It may also improve security and quality of service [81].

To perform NC, encoding and decoding schemes have to be employed on source, intermediate and/or destination nodes. There are several different schemes (with their own variations), but our main focus is on the most common: Exclusive-OR Network Coding (XOR), Linear Network Coding (LNC) and Random Linear Network Coding (RLNC).

2.1.1 Max-Flow Min-Cut Theorem

The Max-Flow Min-Cut Theorem [24], firstly proven by Menger [70], states that the maximum multicast throughput of a network, per channel use, is bounded to the smallest edge cut separating the source and any of the destinations. This minimal cut represents the number of pairwise edge-disjoint paths from source to destination. Such a collection of pairwise-disjoint paths can be found using the Ford-Fulkerson algorithm [40].

In what concerns a unicast network, i.e., single source and destination nodes, the maximum throughput (equivalent to the smallest edge cut), per channel use, can be achieved simply with routing, by sending one packet along each of the edge-disjoint paths [71].

In the case of a multicast network, with a single source and a set of destination nodes, the maximum multicast throughput, per channel use, is not always achieved with routing [71]. However, it is achievable with NC [24]. Moreover, it is achievable with LNC, if the packet alphabet over a finite field is sufficiently large [62].

This is the main theorem of network multicasting. The next section provides a classical introductory example for this theorem and the concept of NC.

2.1.2 Exclusive-OR Network Coding (XOR)

As the name suggests, in Exclusive-OR Network Coding (XOR) [24], the encoding and decoding schemes consist of a bit-wise addition, also referred as Exclusive-OR, or simply, XOR.

In XOR, source nodes transmit (n) original uncoded packets. Intermediate nodes perform encoding by applying a logical XOR on the incoming packets (from a single or multiple source/flow). Decoding is done at destination nodes. In order to decode, these nodes must previously receive and store $(n-1)$ original packets of the coded packet.

To illustrate the concept, let's consider a single source node S, and two destination nodes X and Y. Intermediate nodes C, D, E and V simply multicast incoming packets to the remaining ports. The source node S has two packets, A and B, which must be multicast to X and Y, at the highest rate possible. Also, each directed edge represents a network link, and can transmit only a single packet per time unit. From Figure 2.1 (a), if no coding is allowed, there is a "bottleneck" at the node V. Either the packet A or B is transmitted

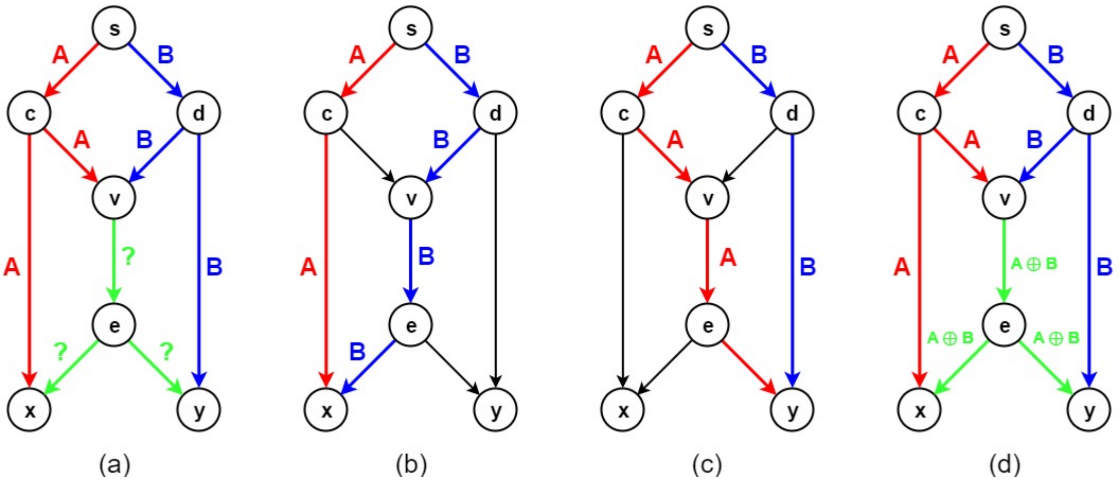


Figure 2.1: Traditional Butterfly Network **(a)** Network bottleneck **(b)** Routing solution to X **(c)** Routing solution to Y **(d)** Network Coding solution.

through the outgoing link. To deliver both packets A and B, to both destinations X and Y, node V requires two time units. The first to transmit packet A, the second to transmit packet B, or vice-versa. This possible routing solution is illustrated in Figure 2.1 (b) and (c). From Figure 2.1 (d), if coding is allowed, by extending node's V functionality from *store-and-forward* to *store-encode-forward*, then it can perform a simple XOR on both packet A and B, and transmit the result. This allows to overcome the congestion of node V, as X obtains A and reconstruct B from A ⊕ (A⊕B) = B and Y obtains B and A = B ⊕ (A⊕B), in a single time unit.

2.1.3 Linear Network Coding (LNC)

In Linear Network Coding (LNC) [62], intermediate nodes transmit to output links, linear combinations of packets from input links. The coefficients of the combination are selected over a finite field F. In particular, the XOR scheme (§ 2.1.2) is the base case of a linear code, where the size of the finite field is just 2. From the literature, the LNC concept [32] can be formally defined as follows.

Consider a network with a single source s and a set of destinations T. Let h be the multicast capacity of the network, and $x_1, \ldots, x_h$ the h packets to multicast from s to T, per unit of time. For each output link e from an intermediate node v, let $y(e)$ denote its carried packet. The packet $y(e)$ is a linear combination of packets $y(e')$ from input links e' of node v, computed as $y(e) = \sum_{e'} \beta_{e'} y(e')$. The coefficients of this linear combination form a vector $\beta_{(e)} = \left[\beta_{e'}(e)\right]$, known as the local encoding vector on edge e. This vector's length equals the number of input links.

Consider $y(e'_1), \ldots, y(e'_h)$ as the original $x_1, \ldots, x_h$. Then, the packet $y(e)$ on any edge e in the network can be computed as a linear combination of the source packets $x_1, \ldots, x_h$, i.e., $y(e) = \sum_{i=1}^{h} g_i(e)x_i$. The coefficients of this linear combination form a vector $g(e) = \big[g_1(e), \ldots, g_h(e)\big]$, known as the global encoding vector on edge e. The global encoding vector $g(e)$ represents the packet $y(e)$ in terms of the source packets $x_1, \ldots, x_h$. Concluding, the global encoding vectors can be computed recursively as $g(e) = \sum_{e'} \beta_{e'}(e)g(e')$, by using the coefficients of the local encoding vectors $\beta(e)$.

Supposing that a destination node $t \in T$ receives the packets $y(e_1), \ldots, y(e_h)$ on input links $e_1, \ldots, e_h$, they can be expressed in terms of the original packets as:

$$\begin{bmatrix} y(e_1) \\ \vdots \\ y(e_h) \end{bmatrix} = \begin{bmatrix} g_1(e_1) & \ldots & g_h(e_1) \\ \vdots & \ddots & \vdots \\ g_1(e_h) & \ldots & g_h(e_h) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_h \end{bmatrix} = G_t \begin{bmatrix} x_1 \\ \vdots \\ x_h \end{bmatrix}$$

where the i_{th} row of the matrix G_t is the global encoding vector associated with edge e_i entering destination t. A network coding solution is found when the matrix G_t has full rank (in this case, h entries). The h original packets are then obtained by inverting the matrix G_t and applying the result to the previously received encoded packets.

Establishing the necessary local encoding vectors (for each node) to find a network coding solution is not trivial. Over the years, many algorithms appeared with that purpose. The first one was proposed by Li et al. [62], but the running time was exponential. The first polynomial time algorithm was proposed by Jaggi and Sander et al. [49]. Other polynomial time solutions are [43, 59]. As of today, the best algorithm is considered to be [59]. Koetter and Médard [54, 53] simplified the problem of finding these local encoding vectors, by alternatively giving an algebraic framework for LNC.

2.1.4 Random Linear Network Coding (RLNC)

In Random Linear Network Coding (RLNC) [47], the local encoding vectors are simply chosen completely at random, over a large enough galois field (GF). If the field is sufficiently large, the resulting code will, with very high probability, achieve the maximum multicast rate for the considered network. The main advantage of RLNC is that the assignment of these vectors can be done in a decentralized way, without the need to run the polynomial time algorithms (as referred in § 2.1.3) to establish them over all network nodes a priori, during startup. This can be useful in practical applications [46, 28].

In this section, we describe, in a less formal and more practical way, the most standard RLNC [62, 47] strategy. There are several variants (§ 2.1.5).

In RLNC, a message (e.g., video, audio, etc) to be sent is broken into several chunks of equal length - each known as a "generation" of the original message. Each generation is further divided into "symbols", also of equal length each. The number of symbols represents the generation size. Each symbol is further divided into elements of x bits, given the considered Galois Field (2^x).

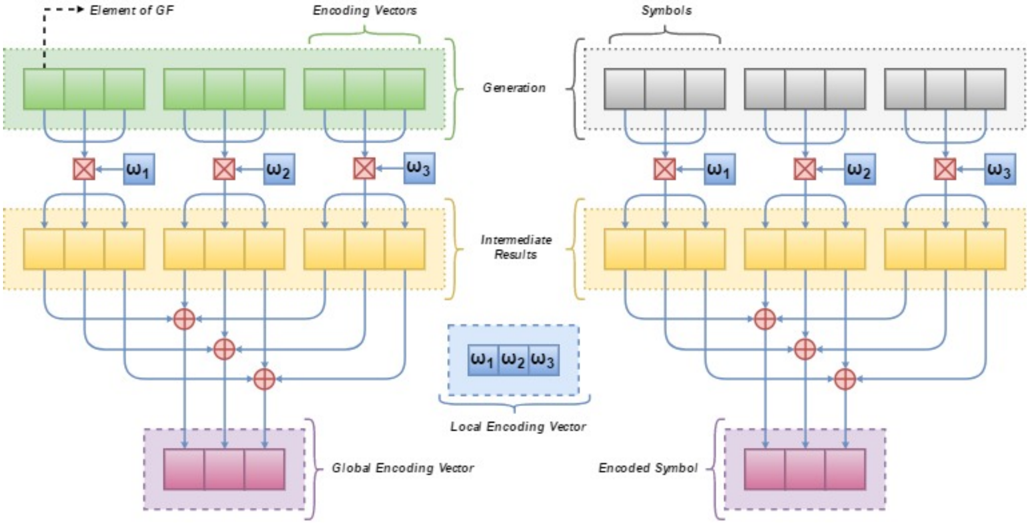


Figure 2.2: Random Linear Network Coding - Encoding.

Encoded symbols are obtained by performing linear combinations of all the original symbols of that generation. For each original symbol, a coefficient is randomly chosen over the considered Galois Field (GF). Then, each element of the given symbol is multiplied by this coefficient. Finally, each element composing the encoded symbol is the result of the addition of all the multiplications performed over the elements of each original symbol by the respective coefficient, of the same position (see Fig. 2.2).

The set of coefficients, used during the encoding process, is known as the local encoding vector. The set of local encoding vectors applied at each network node, from source to destination, forms the global encoding vector.

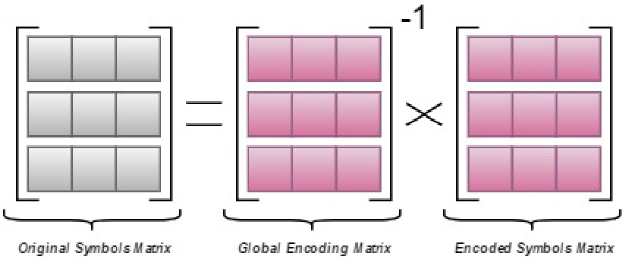


Figure 2.3: Random Linear Network Coding - Decoding.

To decode a generation, a number of linearly independent symbols, equal or greater than the generation size, and the respective global encoding vectors, is required. The symbols are placed on an Encoded Symbols Matrix (R) and the respective global encoding vectors on a Global Encoding Matrix (G), until the matrix G is square and has a rank equal to the generation size. The matrix with the original symbols (O) is obtained by solving $O = G^{-1} \times R$. The inverse of the matrix G is obtained with the Guass-Jordan elimination algorithm, which is a variant of Gaussian elimination (see Fig. 2.3).

2.1.5 Other RLNC Schemes and Variants

As previously mentioned, there are several RLNC strategies. Depending on the use case, one strategy might be more adequate than another. Consequently, no strategy can be defined as the best. Below is a summary of the most well known and commonly used:

- **Standard RLNC**: For a given generation, all symbols are uniformly and randomly combined. Because the symbols are mixed as much as possible, this coding type is typically "dense" and computationally "heavy". As such, for a fairly efficient use, small field sizes are a requirement.

- **Sparse RLNC with uniform density** [38]: Same as the standard RLNC case, with the difference that some symbols of the given generation have a certain probability of being excluded during the encoding process. This is useful when the generation size is substantially high. As a result, the density of the code is reduced, without any downside, and the decoding performance is greatly increased.

- **Sparse RLNC with fixed density** [68]: Given the current generation, only a fixed number of symbols is used as input to the encoding process. If this fixed number of symbols equals the size of the generation, then it is the same as the standard RLNC. This strategy is useful if the decoder is able to provide feedback about the state of the decoding of the current generation, as the encoding/recoding processes can be adjusted accordingly to the current needs and, thus, provide a slight increase in performance.

- **Seed-based RLNC** [31]: Typically, the encoding vector is fully sent alongside the encoded symbol. To reduce this overhead, a random seed, able to generate the encoding vector, is sent instead. This strategy makes recoding more difficult and complex, even impossible in some cases. As such, this is used in scenarios where recoding is very little used, or not used at all.

- **On-the-fly RLNC** [35]: This scheme differs from the traditional block codes. Contrary to these, it is not necessary for all the data to be available before encoding/recoding/decoding can take place. Given the current generation, a symbol can be immediately encoded/recoded as soon as it is available. Decoding is similar. For each newly received symbol, the decoder immediately performs one additional decoding step and releases any resulting decoded symbols, if possible. This coding strategy works well with low-delay services, such as VoIP, video streaming and messaging.

- **Perpetual RLNC** [44]: This is a sparse and well structured coding strategy. The non-zero coding coefficients, i.e., the coding coefficients associated to symbols used on the encoding/recoding process, have a specific location in the coding vector. The width of the non-zero subset of coefficients is analogous to the density of random sparse codes. Due to this characteristic, this approach allows the decoding process to be well structured, which permits achieving higher throughput values than other random sparse codes, especially with high generation sizes.

- **Fulcrum RLNC** [66]: Consists of a concatenated "outer" and "inner" code structure. These codes provide end-to-end performance similar to other large field network codes for high-end receivers, while simultaneously being able to serve low-end nodes that may only, for instance, be able to decode in GF(2).

2.1.6 Advantages and Use Cases

The number of scenarios and use cases where NC might bring benefits are enormous. This section's objective is not to enumerate all of them. Nor is it to formally prove the advantages of NC. Instead its goal is to highlight its benefits and applicability.

As already stated at the beginning of this chapter, NC has successfully been proven to provide reliability and/or to increase (even maximize) throughput of different communication networks. Another benefit is the fact that for certain codes, like linear ones, network nodes don't have the requirement of receiving specific data packets in their totality. Instead, they just have to receive a sufficient number of linearly independent encoded packets to be able to decode and retrieve the original data.

Just like Erasure Correcting Codes (ECC), NC can be used in Point-to-Point Communication Networks with lossy links, to deal with packet losses. If a fairly accurate estimate of the packet loss probability is known, source nodes may pro-actively send extra redundant packets to compensate losses. This technique is known as Forward Error Correction (FEC). Another approach is to send the extra redundant packets retro-actively, which requires the destination to acknowledge the source about the lost packets. In this case, NC simplifies the acknowledgment mechanism, because the destination just has to provide information about the number of lost packets, and not specifically which ones.

Another use case that benefits from NC is reliable multicast, over wireless networks, with lossy links. If no losses were to occur and each channel had ideal bandwidth, all multicast packets would be received by all destination nodes. On the other hand, in the presence of packet losses, reliability can be assured by a retransmission mechanism. Destination nodes just need to request the missing packets from the source. This means that all lost packets are transmitted again. If the several losses, among all destination nodes, are not correlated, then most of the retransmitted packets will not be useful to specific destinations. Basically, due to the uncorrelated nature of packet losses, each destination usually has a different set of received packets. In the end, network resources are wasted and a single retransmission will most likely serve a single destination node. However, with the use of NC, a single retransmission of an encoded packet, instead of a original one, can serve multiple destination nodes. In this case, the source node can send a linear combination of the original data. Each encoded packet might therefore be useful at different destination nodes during the decoding process, and thus serve several nodes simultaneously, instead of just one.

In best-effort multicast networks, reliability is usually not a requirement neither effectively possible. Considering a video streaming to several destination nodes over a wireless network, where acknowledgments are impractical, NC can help to maximize the impact of each retransmission. In the case of linear codes, reliability is ensured by transmitted a certain number of redundant packets for each generation. The redundancy ratio should be tuned according the estimated Packet Error Probability (PER).

Networks characterized by dynamic nodes, i.e. nodes constantly changing their physical position; or connected in a multi-hop fashion, i.e. where some nodes can be exclusively reached over multiple hops, greatly benefit from NC due to its recoding feature. Basically, with LNC schemes, intermediate nodes can create new recoded packets and thus new linear combinations of previously received and stored packets. Recoded packets can be generated from partially decoded generations. As in these networks, source and destination nodes are several hops away, if the recoding scheme at the several intermediate nodes follows an *interflow* scheme, then the same recoded packets might be used by several different destination nodes, to retrieve original data from different flows (i.e., the desired flow by the given destination node).

Additionally, in multi-hop networks a single node has limited (to none) information about other nodes' state, especially if these nodes are several hops away. LNC requires coordination among all nodes as the statically set of coefficient used must guarantee that the final set of packets received at destination nodes is decodable. RLNC provides an implicit solution for this coordination problem as coefficients are randomly chosen over a significant large finite field, which ensures that the final set of packets received by destination nodes is decodable with very high probability.

2.1.7 Network Coding Applications

Despite being a fairly recent topic, NC has had its fair share of applicability. This is easily proven by consulting just a few surveys [32, 36, 25, 37]. Nonetheless, in this section a few applications, that served as a learning base for NC, and inspiration for the design of our solution, are summarized.

COPE

COPE [51] is a packet forwarding architecture for wireless mesh networks that significantly improves its overall throughput. It is based on two key principles: (1) employment of network coding and (2) exploit of the broadcast nature of wireless channels instead of point-to-point link abstractions. This permits taking advantage of coding opportunities to forward several packets in a single transmission. These coding opportunities are detected by a coding layer inserted between the IP and MAC layers. It uses inter-session network coding – packets with different destinations are encoded together. The coding scheme consists of a simple XOR of multiple packets.

COPE incorporates three main techniques:

- **Opportunistic Listening**: Each node is required to store packets in a packet pool to be able to perform coding and decoding. The packet pool consists of two kinds of packets. Packets the node itself has to broadcast, and packets it has overheard. Since wireless is a broadcast medium, there are many opportunities for nodes to overhear transmitted packets. To avoid having nodes discarding overheard packets not meant to them, COPE sets all nodes to promiscuous mode. In this mode, all overheard packets are stored for a limited period of time T. Each node informs its neighbors about its stored packets by broadcasting reception reports.

- **Opportunistic Coding**: Nodes should maximize the number of native packets delivered in a single transmission, while ensuring that each intended next hop has enough information, i.e., all but the native packet, to decode its native packet. This aims to maximize the benefits of coding [51].

- **Learning Neighbor State**: Nodes ideally would need an accurate global view of the network, i.e., to know which packets each neighbor has stored. This information is given from reception reports. However, if the network is severely congested, reception reports may arrive too late or get lost, and so, the solution is to make well-informed guesses. This process is as follows: (1) compute the delivery probability of all paths, (2) broadcast the results to all nodes and (3) use them in a link-state routing protocol to compute shortest paths. An incorrect guess may be made sometimes, making the native packet un-decodable at the destination. Whenever that happens, the native packet is retransmitted again, potentially encoded with a different set of packets.

In COPE, each node maintains four data structures: a packet pool to store overheard packets; a FIFO output queue of packets to be forwarded; two per-neighbor virtual queues (one for small and other for large packets); and an hash table indicating for each packet the probability of each neighbor having that packet. Packets contain three blocks of data: IDs of the encoded native packets, reception reports, and set of acknowledgments to be delivered.

COPE never delays packets in order to encode. If no coding opportunity is available, an uncoded packet is sent. Also, packets headed for the same next hop are never encoded together. Preference to XOR-ing packets of similar lengths is given.

The 802.11 broadcast is not used because it lacks reliability, backoff and retransmission mechanism. Instead, COPE relies on a pseudo-broadcast which unicasts packets that are meant for broadcast. The unicasted packet has its MAC address set to one of the nexthops and a XOR-header listing all nexthops of the packet. Nodes that overhear the packet check this header to see if it is a nexthop.

Although COPE works well, it has some limitations and a more efficient architecture is proposed in [50].

MORE

MORE [28, 29] is a wireless routing protocol for wireless mesh networks. It is also an opportunistic routing protocol, which makes it more reliable against packet losses. Just like COPE [51], MORE detects coding opportunities via a coding layer inserted between the IP and MAC layers. Differently from COPE, MORE employs intra-session network coding – packets with the same destination are encoded together. The coding scheme consists on RLNC.

In this protocol, the rule of each type of node is summarized as follows:

- **Source**: The source partitions the file to be transmitted into batches of K native packets. Then creates a random linear combination of the K native packets of the current batch, and broadcasts the resulting coded packet. Attached to the packet is the coefficient vector used for encoding, the ID of the batch, the source and destination IP addresses and a forwarder list with eligible nodes to forward the packet. The list includes nodes closer to the destination than the source, ordered by their proximity (computed using the ETX metric [34]) to the destination. The source continues to broadcast coded packets of the current batch until the batch is acknowledged by the destination.

- **Forwarders**: Forwarders nodes listen for packets on all transmissions. When a new packet is heard, the node first checks if it is in the packet's forwarder list. If so, the node performs two actions. Firstly, it decodes the packet, via Gaussian Elimination [53, 54], to check if it is an innovative packet – linear independent from previously received packets for the current batch. Innovative packets are stored and non-innovative are discarded. Secondly, it creates a new random linear combination of all packets heard for the current batch and broadcasts the resulting coded packet.

- **Destination**: The destination stores innovative packets and discards non-innovative ones. Once K innovative packets are received for the current batch, it decodes the whole batch by using a matrix inversion technique. It then sends an acknowledgement back to the source, so that it can start sending packets from the next batch.

Avalanche

Avalanche [42] is a peer-to-peer system for file distribution, developed by Microsoft, based on a RLNC coding scheme. An important advantage is that it does not require the knowledge of the underlying topology or any centralized scheduling. In addition, it is robust to extreme situations, such as sudden server and nodes departures and has better performance compared to source coding or no encoding schemes.

Avalanche has a single centralized server, which stores the original files. The server is constantly dividing each file into K blocks and uploading the blocks to different client nodes, at random. To retrieve the original file, the clients exchange blocks only with a small subset of other clients – the neighborhood. The neighborhood is constructed when the client first joins the network and contacts the server to get a list of random clients in the system.

In Avalanche, there are three modes of content propagation: no coding, source coding and network coding. In the case of no coding and source coding, the algorithm for deciding which block to transfer is based exclusively on local information. The block to transfer may be picked among the rarest in the neighborhood, the rarest system-wise among the neighborhood (not practical in large networks), or completely at random.

In the case of network coding, the node generates and sends a random linear combination of all the encoded packets it has received so far, for the given file. Reconstructing the original file is done after receiving K linear independent blocks, associated with the respective coefficient coding vector. With network coding, a client does not need to request specific blocks. Instead, it keeps downloading coded blocks from its neighbors, until it can reconstruct the original file.

Avalanche also offers two incentive mechanisms to discourage free-riding, i.e., having clients take advantage of the network to obtain the desired files without contributing with uploads to other clients. The first is to allocate more upload bandwidth to neighbors from which the client is also downloading, than to those which it is not. The second (tit-for-tat approach, from BitTorrent) is not to upload to a neighbor until the absolute difference of uploading minus downloading from one to another is bounded.

Other Applications

We close this section with references to a few other interesting applications.

Ncos [64] is a framework for realizing network coding over SDN. The paper discusses the possibility of using an XOR network coding scheme through the extension of the OpenFlow protocol [69, 14] and of the functionality of switches.

KODO [73, 45] is an open source C++ network coding library intended to be used in practical studies of network coding algorithms.

Other areas of NC application include Commercial mobile platforms (e.g., PictureViewer [74]), LTE Networks (e.g., Raptor codes in eMBMS [83]), Mobile *Ad Hoc* Networks (e.g., CONCERTO [39]), network security [81, 80], etc.

Listing all the fields and relevant applications of network coding is a never ending task. Therefore, a read on existing surveys [32, 36, 25, 37] is recommended.

2.2 SDN and P4 - Overview

P4 is a declarative language for data plane programming, which works alongside SDN control protocols, such as Openflow. This section provides a brief walktrough over the issues related to traditional computing network equipments, that SDN and Openflow confront and try to solve, which consequently led to the appearance of P4. Finally, the P4 language is explained in detail.

2.2.1 Control Plane Programmability and SDN

In traditional computer networks, the functionality of each node can be divided in three planes:

- **Data plane**: Network nodes that forward packets to the next hop based on control plane decisions.
- **Control plane**: Protocols that populate the forwarding tables of data plane nodes.
- **Management plane**: Services that monitor and configure the control plane.

The problem with traditional architectures, is that these planes are vertically integrated. In particular, the control and data planes are tightly coupled inside vendor-specific closed hardware and software. This results in networks that are complex and hard to manage [56]. In order to change and adapt the existing protocols or even to add new ones, network operators need to configure each network node individually, one-by-one, or use complex low-level scripts to do so. This is the main reason why the innovation of networking infrastructure has progressed at a very slow pace.

Software-Defined Networking (SDN) [56, 77] is a new network paradigm that emerged and aims to solve the shortcomings of more traditional architectures. The Open Networking Foundation [12], defines the SDN architecture as:

- **Directly programmable**: Network control is directly programmable because it is decoupled from forwarding functions.
- **Agile**: Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- **Centrally managed**: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
- **Programmatically configured**: SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
- **Open standards-based and vendor-neutral**: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

Figure 2.4, shows the SDN architecture in more detail. The lowest layer is the infrastructure layer, also called the data plane. It consists of the forwarding devices. At the middle, there is the control layer, also called the control plane. It is responsible for programming and managing the data plane. To that end, it makes use of the information provided by the data plane. It consists of one or more controllers. At the top, there is the application layer, also called the management plane, which contains network applications that assist the control layer in several tasks, such as security, management, monitoring or configuration of the network. These applications can make use of the global view of the network from the controllers.

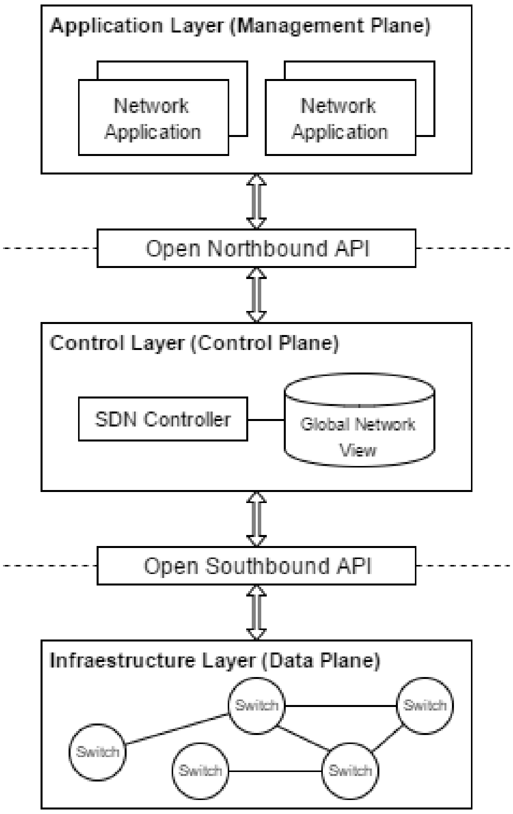


Figure 2.4: Example of SDN architecture.

The controller offers two software interfaces to communicate with network application and forwarding devices, referred as the northbound API and southbound API, respectively. The northbound API is not standardized yet and there are many SDN programming languages that fulfill its purpose, such as: Frenetic [41], Pyretic [78] and Procera [84]. On the other hand, for the southbound API, there is a well-known standard – OpenFlow [69, 14].

2.2.2 OpenFlow, Switch Chips and Motivation for P4

OpenFlow [69, 14] was developed with the objective of having a unique way for software control planes to remotely control OpenFlow-compliant switches from a variety of different vendors. The idea is to make the task of writing better control planes easier. In a great majority of computer networks, switches run the same protocols: Ethernet, IPv4, Access Control Lists (ACLs), VLANs, etc. OpenFlow defines a standard, i.e., an open interface to populate the forwarding tables in these switches. For instance, the hash tables for Ethernet address lookup, the longest-prefix match tables for IPv4 and the wildcard lookups for ACLs. At first, OpenFlow didn't please some vendors as it commoditized their switching products. However, the interest in it has grown tremendously since its conception and it has proven to be a huge success. As of today, there are hundreds of OpenFlow-compliant switches [22, 11, 7, 5, 4, 3, 2] and this architecture has been deployed in enterprise environments [23, 10], service providers [13], and data centers [9, 8].

Still, OpenFlow has a shortcoming. It assumes that switches have a fixed, well-known behavior, typically described in the datasheet of the switch ASIC. Indeed, traditional high-performance switching chips support a fixed set of protocols because they directly implement IEEE and IETF standard protocols in silicon. Changing existing protocols or adding new ones, to measure and control the datapath in a different way, is not possible. Nowadays, it takes about four years to add a new protocol to a fixed-function ASIC. At first, OpenFlow only supported the addition and removal of forwarding entries for four common protocols: Ethernet, VLANs, IPv4 and ACLs. More header-types were added eventually, such as: IPv6, MPLS and VXLAN. Currently, there is support for more than fifty different protocols. Switch vendors can specify which headers the control plane supports, by using the Table Type Patterns (TTP) standard from the Open Networking Foundation (ONF).

In conclusion, OpenFlow doesn't really control the switch functionality. It just provides a mean to populate a set of well-known forwarding tables. An important drawback of using existing switches is their unnecessary complexity and insufficient flexibility. Existing chips are designed to support a huge set of features but only a small subset is usually used. Also, the processing pipelines may be too long. These pipelines contain a fixed number of tables, of fixed size, in a fixed arrangement, and they process a fixed set of headers. The combination of a long fixed pipeline and a large set of features results in resource waste and failure in adapting to new protocols.

To allow switches to expose more of their capabilities to the controller, it is required that more headers and multiple stages of rule tables are added to the OpenFlow specification. This trend can be problematic and unpractical, and it exists because switching chips are not programmable.

If switches offered more support of flexible mechanisms, for parsing packets and matching header fields, thus allowing controller applications to take advantage of these capabilities through a common open interface, a protocol like OpenFlow would not be necessary. Programmers could just define the desired API to create and populate custom tables, as well as define how to process packets.

In fact, reconfigurable (programmable) switch chips already exist for some time (for instance, NPUs and FPGAs). However, their performance is orders of magnitude worse than fixed-function ASICs. Fortunately, recent works in chip design demonstrate that the support to flexible mechanisms, like the ones mentioned above, can be achieved in custom ASICs at terabit speeds [55, 27, 6]. Such programmable switches already exist commercially. Barefoot Tofino [1], for instance, is the first fully programmable switch, and it's also the world's fastest, running at 6.4 Tbps. The problem is that programming these chips is difficult, as each chip has its own low-level interface, akin to microcode programming [26].

This was the motivation to create a common language to program every kind of switch the same way, and thus achieve interoperability between them. This language is known as: Programming Protocol-independent Packet Processors (P4) [26]. Although both OpenFlow and P4 focus on “opening” the forwarding plane, P4 addresses the need to *program* the data plane.

2.2.3 Data Plane Programmability and P4

P4 [26] is a prominent high-level declarative language, which works alongside SDN control protocols, such as OpenFlow. As of the writing of this dissertation, there are two versions of the language: P4-14 [15] and P4-16 [16]. Each one has its own specification and differs greatly both in syntax, semantics and underlying constructs. On the context of this dissertation, P4-14 was used, as it is currently the most stable version. Thus, we refer to P4-14, from now on. This section specifies all the necessary details about P4-14 that were used in, and are needed to understand, our implementation.

Overview

P4 has three main design goals, which also turn out to be its main properties:

- **Reconfigurability**: Packet parsing and processing can be redefined once the switches are deployed.
- **Protocol independence**: Switches are not tied to specific network protocols and packet formats.
- **Device independence**: Packet parsing and functionality is described independently of the details, and specifics, of the underlying hardware. Instead, by taking the switch’s capabilities into consideration, a compiler is used to configure the switch. Basically, it turns a target-independent description (written in P4) into a target-dependent program (used for switch configuration).

P4 expresses how the data plane of a programmable forwarding network device (i.e, router, hardware or software switch, network interface card [NIC], or network function appliance) processes incoming packets. Such P4 compliant devices are classified with the generic term: *target*. According to the specification, a target may directly execute a P4 program but it is assumed that the program is first compiled into a suitable configuration for the target.

Contrary to the data plane, P4 cannot be used to express the control plane of the target. However, in part, it also defines the communication interface between the control plane and data plane.

In traditional fixed-function devices, the data plane and control plane functionalities are defined by the manufacturer. Data plane manipulation is assured by the control plane via table entry management (e.g., routing tables), object configuration (e.g., meters and counters), and control-packets processing (e.g., routing protocol packets) or asynchronous event handling (e.g., link state changes or learning notifications). A P4 compliant target differs from a traditional fixed-function device in two distinct ways. Firstly, it has no built-in knowledge of existing network protocols, as its functionality is not fixed a priori but is defined by the P4 program instead. Secondly, the set of tables and other objects in the data plane are also not fixed and are defined by the P4 program. Hence, control plane and data plane communication is similar but the necessary API, for that purpose, is generated by a P4 compiler.

As such, P4 permits the expression of a broad variety of data plane protocols and behaviors, and thus can be said to be protocol independent. In sum, a P4 program specifies and consists of:

- **Headers**: A header definition describes the sequence and structure of a series of fields. It includes specification of field widths and constraints on field values.
- **Parsers**: A parser definition specifies how to identify headers and valid header sequences within packets.
- **Tables**: Match+action tables are the mechanism for performing packet processing. The P4 program defines the fields on which a table may match and the actions it may execute.
- **Actions**: A complex action, consisting of a set of simpler protocol-independent primitives. These complex actions are available within match+action tables.
- **Pipeline layout and control flow**: The control program determines the table layout within the data plane pipeline and the packet flow through the pipeline of match+action tables.

P4 addresses the configuration of a forwarding network device. After configuration, tables are populated and packet processing begins. Reconfiguration can still be performed during packet processing, usually by recurring to a runtime CLI interface, exposed by the target.

P4 Targets and Compilers

The P4 Consortium [20] provides a GitHub repository with an implementation of a P4 software switch, also known as Behavioral Model 2 [18] (nicknamed **bmv2**). It offers almost complete support for both P4-14 and P4-16. By default, it is prepared for the P4-14 architecture (i.e., Abstract Forwarding Model). The runnable components themselves (i.e., the runnable P4 switches), are within a directory named **targets**. There are three distinct targets: `simple_router`, `l2_switch` and `simple_switch`. Each of these targets implements, in software, the P4-14 specification to a different extent. For instance, *simple_router* and *l2_switch* are very limited because they implement exclusively a handful of P4 primitives. On the contrary, *simple_switch* is the most complete, thus being the reference target for P4. Therefore, *simple_switch* is the one we ended up using. For the remaining of this dissertation we refer to *simple_switch*, whenever bmv2 is mentioned.

To map a P4 program onto the target switch's specific hardware or software platform, we need a compiler. Particularly, P4-14 programs can be compiled to *simple_switch* with **p4c** [19], which is the official, yet still alpha, P4 compiler.

The Abstract Forwarding Model

P4 is based on an abstract forwarding model, which generalizes packet processing in different forwarding devices (e.g., Ethernet switches, load balancers, routers) and by different technologies (e.g., fixed-function ASICs, NPUs, reconfigurable switches, software switches, FPGAs). It consists of a parser and a set of match-action ingress and egress table resources. The parser identifies and extracts headers (and respective fields within) of each incoming packet, for later manipulation. A match-action table performs a lookup on a subset of header fields and, for the first match, applies the respective action.

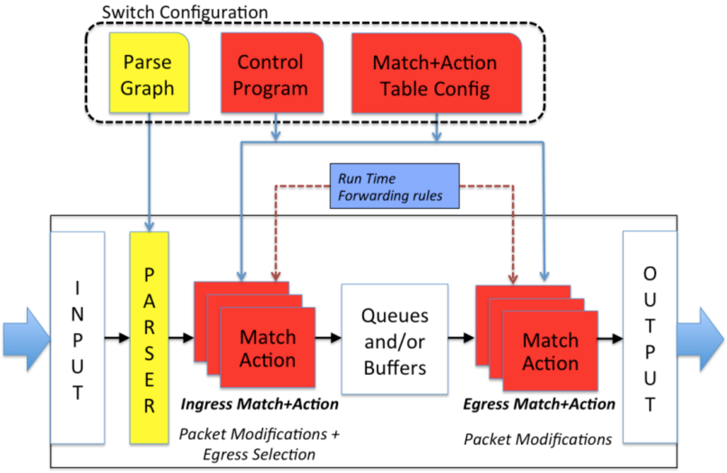


Figure 2.5: Abstract Forwarding Model [15].

Figure 2.5 shows a high-level representation of the P4 abstract forwarding model. The current "widely supported" P4-14 Specification [15], defines that a target must follow a set of rules, in order to operate:

- For each packet, the parser produces a *Parsed Representation* on which match+action tables operate.
- The match+action tables in the *Ingress Pipeline* generate an *Egress Specification* which determines the set of ports (and number of packet instances for each port) to which the packet will be sent.
- The *Queuing Mechanism* processes the *Egress Specification*, generates the necessary packet instances and submits each of them to the *Egress Pipeline*. Oversubscription for an output port may cause the Egress queuing to buffer packets, although this is not mandated by P4.
- A packet instance's physical destination is determined before entering the *Egress Pipeline*. Once in the *Egress Pipeline*, the destination is assumed not to change (though the packet may be dropped or its headers further modified).
- After all processing by the *Egress Pipeline* is complete, the packet instance's header is formed from the *Parsed Representation* (as modified by match+action processing) and the resulting packet is transmitted.

P4 focuses on the specification of the parser, match+action tables and the control flow through the pipelines. Programmers control this by writing a P4 program which specifies the switch configuration as shown at the top of Figure 2.5.

The *Queuing Mechanism* functionality and the *Egress Specification* semantics are not present in the most recent version of the P4 specification. Packet cloning and recirculation are supported features also.

P4 Abstractions

The P4 language provides several abstractions. Together, the set of instances of each, defines a P4 program:

- **Header type**: A specification of the set of fields, and respective sizes, of headers found within a packet.
- **Header instance**: A specific instance of a packet header or metadata.
- **Parser state function**: Describe the permitted sequences of headers, and how to properly identify and extract the headers and fields, within each received packet.

- **Action function**: Set of primitive actions that modify header fields, metadata and registers.
- **Table instance**: Specify the set of matching header fields and the permitted action functions within the given table.
- **Control flow function**: Imperative description of the match+tables to be applied, and in which order. The order might be influenced by logical conditions.
- **Stateful memories**: Data structures (counters, meters and registers) which persist across each packet processing.

Adding to the high level abstractions above, the following are used for a header instance:

- **Metadata**: Per-packet state which may or may not be derived from packet data, stateful memories and/or action primitives.
- **Header stack**: An array of header instances.
- **Dependent fields**: Fields whose values depend on a calculation applied to other fields or constants.

Parser state functions can also make use of:

- **Value set**: Values updatable at runtime and used to determine parser state transitions.
- **Checksums**: The ability to apply a function to a set of bytes from the packet and test that a field matches the calculation.

Headers and Fields

Header types describe an ordered layout of field names, alongside the respective bit-width, and are used to declare header and metadata instances.

An example declaration for a MPLS header is:

```
header_type mpls_t {
  fields {
    label : 20;  // width in bits
    exp : 3;
    bos : 1;
    ttl : 8;
  }
}
```

A packet may contain multiple instances of a given header type. There are two types of instances: packet headers and metadata. A header example is:

```
header   mpls_t   packet_type;
```

This indicates that space should be allocated in the *Parsed Representation* of the packet for a *mpls_t* header. It may be referenced during parsing and match+action by the name *packet_type*.

A metadata example is:

```
metadata   local_metadata_t   local_metadata;
```

This indicates that a *local_metadata_t* type object called *local_metadata* should be allocated for reference during match+action.

P4 supports the notion of a header stack, which is a sequence of adjacent headers of the same type, and is declared as an array. MPLS is the most common example:

```
header   mpls_t   coefficients[100];
```

Header stack instances are referenced using bracket notation and such references are equivalent to a non-stack instance reference.

References to header and metadata instances, and their respective fields, are made for match, action and control flow specifications. Headers are referenced via their instance names, and header stacks with an index in square brackets, and fields with dotted notation. The keyword *last* serves as an index to refer to the largest-index valid instance of a header stack.

Stateless Memory (Metadata)

Metadata represents per-packet state which may not be derived from packet data. It can be read and written by primitive and compound actions. However, some metadata has special significance to the intrinsic target's operation - *Standard Intrinsic Metadata*. There are some Standard Intrinsic Metadata fields which are mandatory and required for P4 compliant targets to operate. Below are some of these fields, defined for the metadata instance *standard_metadata*:

- **ingress_port**: The port on which the packet arrived. Set prior to parsing. Always defined. Read only.

- **egress_spec**: Specification of an egress. Undefined until set by match+action during ingress control processing. This is the intended egress as opposed to the committed physical port(s). May be a physical port, a logical interface or a multicast group.
- **egress_port**: The physical port to which this packet instance is committed. This value is determined by the Buffering Mechanism and so is valid only for egress match+action stages. Read only.

In addition to the *standard_metadata* fields, targets may provide their own definitions of *intrinsic_metadata* to offer more advanced features, although programs which depend on them may not be portable. For example, the P4 reference target (bmv2) defines the following:

```
header_type intrinsic_metadata_t {
  fields {
    ingress_global_timestamp : 48;
    lf_field_list : 32;
    mcast_grp : 16;
    egress_rid : 16;
    resubmit_flag : 8;
    recirculate_flag : 8;
  }
}

metadata  intrinsic_metadata_t  intrinsic_metadata;
```

The *mcast_grp* is required for the multicast feature. For a packet to be multicast, this field needs to be written in the ingress pipeline. A value of 0 means no multicast. This value must be one of a valid multicast group configured through a runtime API.

Here is an example for the reference target (bmv2):

```
mc_mgrp_create 1
mc_node_create 0 2
mc_node_create 1 3
mc_node_associate 1 0
mc_node_associate 1 1
```

The first line creates a multicast group with identifier 1. The second and third lines, fetch logical ports 2 and 3. Finally, the fourth and fifth lines associate the fetched ports to the multicast group.

Stateful Memory (Registers)

Counters, meters and registers are stateful memories, which persist across the processing of each packet. Registers can be read and written in action functions and are generic enough to keep all types of state.

Here is an example of register declaration:

```
register some_array {
  width: 32;
  instance_count: 100;
}
```

As the name suggests, the *width* attribute indicates the bit-width of each register's instance. The *instance_count* attribute indicates the number of register's instances to allocate. A register can only be directly used as input to *register_read* and *register_write* primitives.

However, a register typically holds state that is used/updated on several primitive/compound actions and in if-else control flow logical statements. Therefore, in order to use/update it's state, a register under these circumstances typically has a metadata field associated. This metadata field is used to temporarily hold the register's state and use it as a parameter to other primitive/compound actions or in control flow conditionals.

Parser Specification

P4 models the parser as a state machine. A state can be a simple state transition decision triggered on constants or header/metadata instance field values; or may consist on the extraction of headers. To be more detailed, parsing starts at the packet's first byte, and the packet's current offset is maintained by the parser. Headers are extracted from the packet at the current offset into per-packet header instances (which are marked as valid as well) and the *Parsed Representation* of the packet is updated. The parser then indicates the next valid packet's byte by updating the current offset, and makes a state transition.

Below is an example of a parser state machine. Parsing starts at the *start* state and proceeds until an explicit reference to *ingress* is made, which terminates parsing and invokes the ingress control flow function. As a final result, the parser produces a *Parsed Representation* of the packet, on which the ingress pipeline of match+action tables operate and might update. Each state is represented as a parser function.

A parser function may exit with the *return* statement, specifying the name, to another parser function or control function.

```
parser start {
  return parse_ethernet;
}
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    ETHERTYPE_MPLS : extract_coefficient;
                  default : ingress;
  }
}
parser extract_coefficient {
  extract(coefficients[next]);
  return select(latest.bos) {
         0 : extract_coefficient;
         1 : ingress;
    default : set_drop_flag;
  }
}
parser set_drop_flag {
  set_metadata(parser_meta.drop_flag, TRUE);
  return ingress;
}
```

The *extract* function takes a header instance as a parameter, copies data from the packet at the current offset into that header instance and moves the current parsing location to the end of that header. The header instance cannot be metadata. For header stacks, the special identifier *next* is used to extract data into the next free header on the stack.

Select is a function that takes a list of fields, separated by commas, and compares them, with the entries defined inside the curly brackets. It allows branching to different states.

The special identifier *latest* is a reference to the most recently extracted header instance with the current parse state function. A reference to *latest* without a preceding *extract* function call results in an error.

The *current* function allows the reference of bits not yet parsed into header/metadata instance fields. It has two arguments. The first is the bit offset from the packet's current offset, and the second is the bit width of the reference.

The specification also defines meanings to deal with errors that might occur during parsing. Some errors are immediately recognized as standard parser exceptions, which may be triggered implicitly. Other errors may be triggered by explicitly defining a *parser_exception* function and call it by using *parse_error* within a parser state function. Packets may be immediately dropped at this stage, by invoking *parser_drop* within a *parser_exception* function.

However, as of the writing of this dissertation, the reference target (bmv2) doesn't support exception handling and packet drops, yet. Therefore, the workaround is to use the *set_metadata* statement to mark the packet to be dropped at the ingress pipeline. The *set_drop_flag* parser state function, from the example above, illustrates this.

Action Specifications

Actions are imperative functions, composed of primitive actions. Here is an example:

```
action action_GF_mult (index1, index2) {
  register_read(meta.log1, GF256_log, index1);
  register_read(meta.log2, GF256_log, index2);

  register_read(meta.invlog, GF256_invlog, meta.log1 + meta.log2);
}
```

The function name is used for table specification and population. Table population is performed by the runtime API, and makes use of the action function's name and parameters to associate it with a given table entry. When a match occurs and that entry is selected, the parameters are passed to the action. The P4 table specifications might be used to generate the runtime APIs with parameters corresponding to the action parameters for the entry's action. The compiler is responsible for ensuring that the mapping of the runtime APIs to the P4 program is consistent. In addition to values from the matching table entry, the action operation has access to headers and metadata in the *Parsed Representation*.

The declaration of an action function must follow some rules. All the defined action parameters are required, thus optional parameters are not allowed. Also, the action may only call primitive actions and/or other action functions. Finally, recursion is not supported. P4 assumes sequential execution of all the primitive actions within an action function.

As previously mentioned, action functions are built from primitive actions. There is a standard set of primitive actions. A target may support additional target-specific primitives, but that may condition portability. Besides being a standard, not all target may support all the standard primitive actions, due to underlying hardware and/or software limitations. Below is a brief summary of the subset of standard primitive actions, used in the context of this dissertation. More detailed documentation can be found in the specification.

- **modify_field**: Sets the value of a field in the packet's *Parsed Representation* or metadata header.
- **modify_field_rng_uniform**: Generates a random number from a given range and store in a field.

- **bit_xor**: Performs a bitwise XOR operation on two values and stores the result in a field.
- **drop**: Drops a packet in the egress pipeline.
- **register_read**: Reads from a register instance index and stores the value into a field.
- **register_write**: Writes a value into a register instance index.

Table Specifications

A table declaration specifies a list of match fields, action operations, and possibly other attributes. Here is an example of a table declaration:

```
table table_mult {
   reads {
      packet_type : valid;
      packet_type.label : exact;
   }
   actions {
      _nop;
      action_GF_mult;
   }
   size : 1;
}
```

The *size* attribute indicates the exact number of entries required for the table. If, at run-time, the table has these many entries, the target may reject further insert operations.

The *reads* attribute specifies the list of match fields used for packet matching. A match field is either a reference to a header/field or the validity bit for a header. The semantic of a match is the logical conjunction (AND) of all the match field specifications. There are essentially two types of matches:

- **exact**: The field value is matched against the table entries. A given entry is selected if identical values are found.
- **valid**: Indicates that the field's header should be tested for validity. Table entries must set the field as valid (value of 1) or invalid (value of 0), as a matching criteria. Header references are exclusively used with this type of match. Header instances may also use it. Metadata instances do not, because they are always valid.

The *actions* attribute indicates which actions are available to this table's entries. Actions may be action functions or primitive actions. If there are no table entry matches, a default action is applied. If the default action isn't specified and there are no table entry matches, the table does not affect the packet. If the *reads* attribute is not declared, the default action is always executed, if specified. Otherwise, the table also does not affect the packet.

Entries are inserted with the runtime API and each rule specifies a single action per entry. As such, for a given packet processing only one action may be executed per table. The runtime table entry insert operation, which is not part of P4, must specify values for each match field in the *reads* attribute; and the name and parameters to the action function, if that's the case.

Here is an example, with the runtime API, for the reference target (bmv2):

```
table_set_default    table_mult    _nop
table_add            table_mult    action_GF_mult    1  666  =>  125  250
```

The first line sets the *table_mult* default action to *_nop*. The second line adds an entry to *table_mult*, which specifies that a MPLS packet header with label 666 must be valid, as a matching criteria; and that passes the values 125 and 250 as parameters to *action_GF_mult*.

Packet Processing and Control Flow

The control flow defines the table application order, to which a packet is subjected after parsing. It is expressed, at configuration time, as an imperative program and may consist of table invocations, calls to other control functions and if-else statements that conditions the control flow. When the ingress function completes, and unless it is discarded, the packet is passed to the *Queuing Mechanism*. When the packet is dequeued, the egress control function is called, if defined. Finally, the packet is forwarded to a specific port, given by the *egress_port* field of the target's *standard_metadata*.

Table executions are invoked on the packet with the *apply* operator, which has three operation modes:

- **Sequential**: The next control flow statement is executed unconditionally.
- **Action Selection**: The block of instructions to execute is determined by the applied action in the table.
- **Match Selection**: The block of instructions to execute is determined by whether a match was found in the table, or not.

The following simple control flow programs, and packet processing pipelines, provide an illustrative example of each:

Sequential Mode

```
control ingress {

  apply(table_1);

  if(meta.index < 10) {
    apply(table_2);
  }
  else {
    apply(table_3);
  }
}
```

Action Selection Mode

```
control ingress {

  apply(table_1) {
    hit {
      apply(table_2);
    }
    miss {
      apply(table_3);
    }
  }
}
```

Match Selection Mode

```
control ingress {

  apply(table_1) {
    table_1_action {
      apply(table_2);
    }
    default {
      apply(table_3);
    }
  }
}
```

Match selection is done by specifying a block enclosed in braces following the apply operation with hit and/or miss as the case selection labels. *Action selection* is done by having action names as the case selection labels. Action selection and match selection modes cannot be intermixed, due to the fact that the match selection uses the reserved words *hit* and *miss*, which cannot be used as function names. The control blocks specified by these two modes can also affect the control flow to which the packet is subject. Only the *sequential* mode was used on the context of this dissertation.

2.2.4 P4 Applications

Being a versatile and prominent language, some relevant P4 programs and related works started to emerge recently [21]. A few interesting works, which served as inspiration and as mean to better understand the programming logic for P4, are listed below.

PISCES

Perhaps the most relevant P4 related work, for this dissertation is: Programmable, Protocol-Independent Software Switch (PISCES) [79]. PISCES is an extension of OpenVSwitch (OVS) [75, 76], which is the most currently used software switch in cloud environments. PISCES allows custom protocol specification, without requiring direct modifications to the switch source code, by using P4. It aims to ease the addition of new protocols and the removal of unused ones. It also achieves greater visibility into the network and allows performing network functions at the switch. As input, PISCES takes a P4 source program that is compiled into C language code. The resulting code is combined with a modified OVS to obtain a custom switch with higher performance. The main challenges faced in the development of PISCES are indeed related with performance. Namely, (1) how to ensure that a significant amount of packets are not diverted to the slow path and (2) how to keep the per-packet processing cost low on the fast path; even in the presence of arbitrary encapsulation and decapsulation of headers, complex control flow, and expensive checksum and validation operations. To address these challenges, PISCES incorporates some optimizations like: inline editing of headers, incremental checksums, clever caching, etc.

UnivMon

UnivMon [65] is another project that uses P4. It is a framework, based on universal sketching, for network monitoring. In UnivMon, the programmer specifies a monitoring task as a single program, written using sketch primitives against a “one-big switch” [65] virtual network. Then, the system breaks the source code of the program into smaller code fragments that sample incoming traffic streams, populate local sketch counters on individual switches, and aggregate counters across switches to obtain the final result. The data plane programs that implement sampling and tabulation are specified using P4.

NetPaxos

NetPaxos [33] is another interesting work that uses the P4 language. In this paper, P4 was used to explore how adding custom headers to packets could simplify the Paxos Consensus Algorithm [58, 30] execution at end nodes of a network.

2.3 Summary

In this chapter, we explored the fundamental aspects on both NC (§ 2.1) and the P4 language (§ 2.2), in the context of this dissertation.

We began by explaining the main multicast related theorem (§ 2.1.1), which motivates the usage of NC; and introduced the XOR scheme (§ 2.1.2) to exemplify this result. Then, we described LNC (§ 2.1.3) and RLNC (§ 2.1.4); and explored some of its variants (§ 2.1.5), advantages and use cases (§ 2.1.6). Finally, we overviewed practical applications of NC (§ 2.1.7). Our main focus was on linear codes (namely, XOR and RLNC) because these are the schemes we considered for the design and implementation of our solution.

For the P4 language, we gave the context (§ 2.2.1) and motivation (§ 2.2.2) for its appearance. Then, we described the language itself (§ 2.2.3) and provided some of its practical applications (§ 2.2.4).

Chapter 3 – Design

In the previous chapter we summarized the fundamentals on both NC and P4. NC is what we pretend to deploy in network switches. We reviewed the basic concept, several coding schemes and examples of its applicability, which served as inspiration for the design of our solution. In particular, § 2.1.2 and § 2.1.4 presented the grounding base of the coding schemes that were considered - XOR and RLNC. With the advent of programmable switching hardware, it became possible to implement non-trivial network functions in real hardware. Furthermore, P4 provides the means to program these coding schemes. We explored the motivation behind the creation of the language, how it works and some related works.

Our goal is to deploy XOR and RLNC coding schemes on a P4 compliant target's data plane pipeline. We refer to our solutions as P4-XOR Switch and P4-RLNC Switch, respectively. As explained in § 2.2.3, packet processing in these devices operates in accordance with an underlying forwarding abstracting model. In a way, this characterizes our solution as an overlying system. System design is the process of defining the architecture, modules, protocols and data components for a system to satisfy specified requirements, which in our case, are *Performance* (in terms of unicast and multicast throughput gains) and *Resilience* (to losses of specific packets, node's failures and departures).

We begin the chapter by proposing an overlying architecture (§ 3.1) for our switches. Following from this architecture, we describe the design of its main modules, in the context of both the P4-XOR Switch (§ 3.2) and the P4-RLNC Switch (§ 3.3).

3.1 Architecture

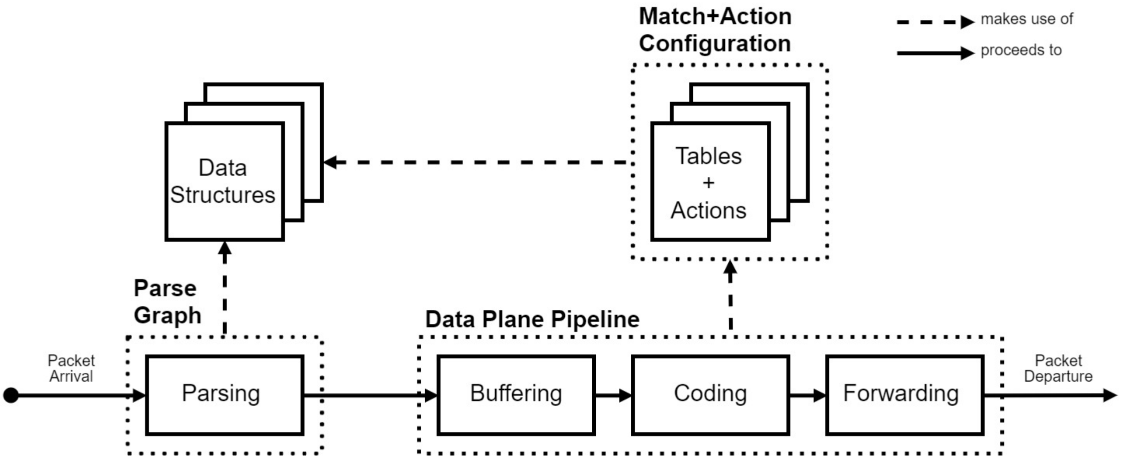


Figure 3.1: Representation of the overlying architecture of the current solution.

The proposed architecture is depicted in figure 3.1. As we already mentioned, it is highly dictated by the underlying forwarding model and the P4 language itself. It consists on the declaration of a parse graph, a data plane pipeline, match+action tables and other auxiliary data structures. The parser is fed the arriving packet and validates, extracts and forwards the necessary information to the data plane ingress pipeline, which performs buffering, encoding and forwarding.

3.2 P4-XOR Switch

The P4-XOR switch deploys an interflow XOR scheme, of two distinct flows. The objective is to encode two packets, one from each flow, with the same sequence number. For each flow, the switch has a buffer, which is used to store the packets for future encoding opportunities. To distinguish each flow and enumerate the packets, we use a custom header. This header also differentiates encoding traffic (involved in the XOR operation) from normal traffic.

On the remaining of this section, we propose a protocol header (§ 3.2.1), which consolidates the necessary information for the coding scheme and the switch's internal functionality as well. We also review the design of the main modules of the overlying architecture, namely the parser (§ 3.2.2), buffering module (§ 3.2.3), coding module (§ 3.2.4) and forwarding module (§ 3.2.5). We close the section by consolidating the switch's operation (§ 3.2.6) in algorithmic form, which provides a more detailed, and in-depth, overview of how to implement our solution in P4-14.

3.2.1 Protocol Header Format

Below is a detailed low-level representation of the XOR protocol header.

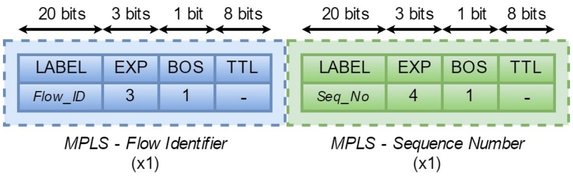


Figure 3.2: P4-XOR Switch's Protocol Header.

To make this header easier to construct at source nodes and parsed at destination nodes, we use an MPLS stack of two headers. The first identifies the packet's flow[1], the second the packet's sequence number[2]. We use a specific header to enumerate packets because sequence numbers of higher-layer protocols (e.g., TCP) may not uniquely identify a packet. For instance, in TCP, sequence numbers of two different connections (i.e., two distinct flows) are completely uncorrelated.

Let's review the MPLS stack in more detail. The LABEL field holds the header information per se. *Flow_ID* represents a value that uniquely identifies the flow; *Seq_No* the sequence number of the packet. The EXP field identifies the type of MPLS header. A value of 3 represents a *MPLS - Flow Identifier* header; a value of 4 a *MPLS - Sequence Number* header. Because this is a stack, the BOS indicates the last header of each type. A value of 0 indicates that the next header is also MPLS and of the same type; a value of 1 indicates that the current header is the last MPLS of its type. In sum, a packet can be viewed as:

- pkt = (pkt.flowID, pkt.seqNo, pkt.payload)

Note that, *pkt.flowID* and *pkt.seqNo* do not necessarily have to be encapsulated within an MPLS header. They could consist of a simple sequence of bits within the packet, where its length and position are fixed and well known among all nodes of the network. However, there would be no way to validate if the values suffered unexpected changes, due to bit flips or a malicious attack during transmission. Thus, we rely on MPLS to be able to make more validations, and the solution more resilient.

Our packet format has no information, distinguishing if the packet is an original or encoded one. We assume that packets with a MPLS header stack are always encoded, and not original. Without this assumption, the packet would need additional information (e.g., a flag of one bit or an additional MPLS header type), to make this distinction.

[1] Referred interchangeably as *pkt.flowID*, from now on.

[2] Referred interchangeably as *pkt.seqNo*, from now on.

Finally, it is important to highlight that this header was defined considering that the P4-XOR Switch encodes together two packets with the same sequence number, over two distinct flows. If more flows were considered, and encoding was not restricted to packets with the same sequence number and/or over all the existing encoding flows, then MPLS header stack would require additional information. Besides its own *pkt.flowID* and *pkt.seqNo*, the packet would need the *pkt.flowID* and *pkt.seqNo* of the other packets to XOR together. For instance, if the packet A with sequence number 4 from flow 1, was to be encoded together with the packet B with sequence number 5 from flow 2, and the packet C with sequence number 6 from flow 3, then the packet A would need to have six MPLS headers with all these information.

3.2.2 Parse Graph

Below is a high-level representation of the P4-XOR switch parser.

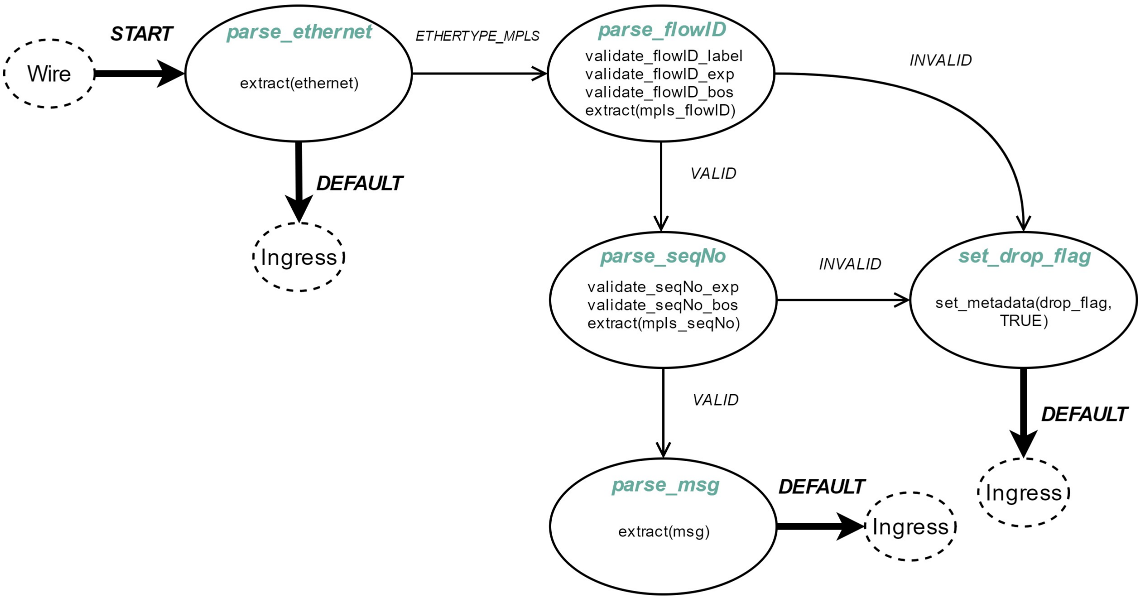


Figure 3.3: High-level representation of the P4-XOR Switch's parser.

Besides the MPLS header stack, the encoding traffic also contains an Ethernet header. First the parser verifies if the packet has a valid Ethernet header, i.e. with an etherType identifying the protocol of the encapsulated frame's payload as MPLS.

If so, it knows that the packet belongs to the encoding traffic, and it proceeds to the validation of the MPLS header stack (according to the format already explained) and the extraction of useful information, i.e. the LABEL fields. If any of the fields violates the proposed format, the packet must be dropped. To drop packets at the parser, an exception that calls a primitive action to drop the packet must be defined and caught. Recall however, that bmv2 doesn't currently support this primitive action (although present in the specification). So, the work around is to use metadata to mark the packet as invalid, so that it can dropped once it reaches the ingress control flow.

Once the MPLS header stack is correctly validated and extracted, the remaining payload[3] is extracted and the packet is sent to the ingress control flow.

Packets without an MPLS header stack are not part of XOR and so, are immediately returned to ingress control flow to be marked for multicast on the remaining ports of the switch. This is the default behavior of our code. In some cases, the code is adjusted to forward to a specific port, instead. More details in § 3.2.5.

3.2.3 Buffering Module

Arriving packets are not encoded alone. Therefore, each one must be previously buffered in order to be immediately available when an encoding opportunity arises. This is why a buffering module is required.

Currently, bmv2 and P4 itself do not offer support for access and control over the queues and/or buffers, between the ingress and egress pipelines, that P4 compliant targets are expected to implement, according to the Abstract Forwarding Model. As such, we had to design our own buffering module (see Fig. 3.4).

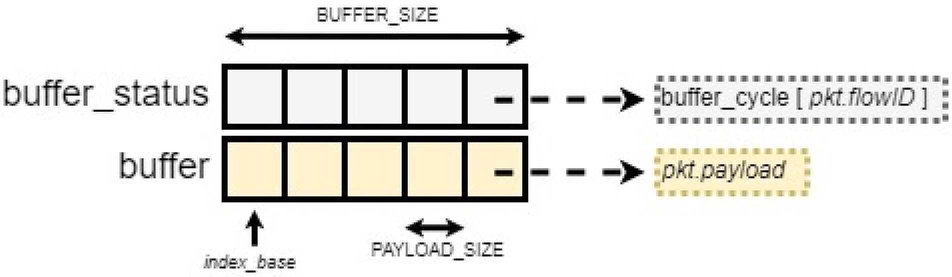


Figure 3.4: P4-XOR Switch's Buffering Model.

For each flow involved in XOR, there is a **buffer**. The buffer itself is an array of registers with BUFFER_SIZE cells. Each cell holds a *pkt.payload*. The chosen buffer for an arriving packet is determined by the *pkt.flowID*. Due to resource constraints, the buffer cannot be infinite and hence, we opted for a circular buffer that can be re-utilized.

In order to implement a circular buffer, we establish an *index_base* for the first cell. Indexation is given by *(index = pkt.seqNo - index_base)*. In practice, *index_base* represents the lower bound of the *pkt.seqNo* that is accepted.

When the computed index is greater than BUFFER_SIZE, the buffer has completed a cycle. Each buffer keeps track of its respective current cycle, on a cell of a **buffer_cycle**

[3] Referred interchangeably as *pkt.payload*, from now on.

array of registers. In the presence of a new cycle, *index_base* is updated as *(index_base = index_base + BUFFER_SIZE)*. The respective cell in the *buffer_cycle* is also updated. This will increase the lower bound.

For the upper bound we settled at *(index_base + 2* × *BUFFER_SIZE)*, but this might be adjusted. As a result, the buffer might have stored *pkt.payloads* from different cycles. Consequently, a complementary buffer indicating the cycle to which the stored *pkt.payload* belongs is needed. Let's call it **buffer_status**. When a *pkt.payload* is buffered, the current *buffer_cycle* is written to the respective cell on *buffer_status*.

It is important to note that the *index* calculation results in negative integers for packets below the lower bound. However, by default P4 considers unsigned integers. So in this case, the result is an *index* much greater than the upper bound, which results in the packet not being buffered, as we expected.

This module, jointly with the Coding Module, is exclusively applied to encoding traffic (meant for XOR encoding). The switch may enable or disable these functionalities with a *nc_enabled_flag* register.

3.2.4 Coding Module

The Coding Module follows from the butterfly network example given in § 2.1.2. This implies that our solution is interflow, as it mixes together *pkt.payloads* with the same *pkt.seqNo*, from two different *pkt.flowIDs*. The encoding/decoding operation consists on arithmetic over GF(2), meaning that is as simple as performing XOR.

For each arriving packet from an encoding flow, first the Buffering Module is applied. Then, we use the previously computed *index* to fetch the *pkt.payloads* from all the *buffers* to metadata. The same for their respective *buffer_status*. Then we compare all the *buffer_status*. If the comparison is TRUE, then the fetched *pkt.payloads* share the same *buffer_cycle*, which entails that their *pkt.seqNo* is the same. In that case, we encode the current *pkt* by updating its *pkt.payload* with an XOR over all the fetched *pkt.payloads* from the buffers.

3.2.5 Forwarding Module

The Forwarding Module is fairly simplistic and not particularly relevant for this work. We defined and changed it, depending on evaluation requirements.

By default, arriving packets without the proposed MPLS header stack are viewed as normal traffic, and are immediately multicasted on the remaining ports of the switch, without being subject to the Buffering and Coding Modules.

Encoding traffic is multicasted by default as well. However, in some cases we opted for forwarding it to a specific port, instead. This decision is based on the incoming port and/or flow. Changing the forwarding decision from multicast to unicast might implicate different matches on the corresponding forwarding table, i.e. changing the code of the table declaration itself. In that case, we can't simply change the default action and repopulate the table, via the bmv2 runtime CLI interface. It doesn't support runtime changes to the P4 code, for obvious reasons. Recompilation of the code and rebooting of the switch, is required.

The reason for the behavior of this module to consist of a multicast on the remaining ports of the switch, or forwarding to a specific port, is simple. The goal is to deploy NC schemes within the data plane of programmable switches. Thus, our focus is on the data plane, not the control plane. Routing management is a responsibility of the control plane. More complex and generic routing strategies, would require a complete overview over the network and/or contemplating existing protocols. As we are not implementing a controller, we cannot provide a complete overview over the network, and update the match-action tables of our switches, at runtime automatically. On top of that, focusing on implementing and dealing with the wide range of existing protocols over several layers (e.g., TCP, UDP, IP, BGP, etc) would get away from, and defeat the purpose of, the objectives of this dissertation.

3.2.6 Operation

So far, we have seen the format of encoding packets; how parsing is performed; and the individual modules of the data plane pipeline. Putting it all together in algorithmic form,

Algorithm 1 Initialization of the P4-XOR switch

```
procedure INITIALIZE_XOR_SWITCH()

    // Load global and parser constants
    TRUE = $s.read$("global_constants.h")                    ▷ $s$ is the switch
    FALSE = $s.read$("global_constants.h")
    PAYLOAD_SIZE = $s.read$("global_constants.h")
    BUFFER_SIZE = $s.read$("global_constants.h")
    ...

    Populate_Tables($cf$)                                    ▷ $cf$ is the configuration file

    // Load some initial state to registers
    $r.nc\_enabled\_flag \leftarrow cf.read()$               ▷ $r$ are the registers of $s$
    $r.buffer\_1\_cycle \leftarrow cf.read()$
    $r.buffer\_2\_cycle \leftarrow cf.read()$
end procedure
```

Algorithm 2 Data Plane of the P4-XOR Switch

```
procedure XOR_DATA_PLANE()
    $m.nc\_enabled\_flag \leftarrow r.nc\_enabled\_flag$                ▷ $m$ are the metadata of $s$
    $m.buffer\_1\_cycle \leftarrow r.buffer\_cycle[1]$                  ▷ $r$ are the registers of $s$
    $m.buffer\_2\_cycle \leftarrow r.buffer\_cycle[2]$
Phase 1 - Compute arriving packet's buffer index
    if $m.nc\_enabled\_flag ==$ TRUE and $pkt$ is $valid$ then
        $m.index\_base \leftarrow r.index\_base[i]$
        $m.index \leftarrow pkt.seqNo - m.index\_base$

        if $m.index >$ (2 $\times$ BUFFER_SIZE - 1) then
            Drop($pkt$)
        else
Phase 1.1 - Update arriving packet's buffer index and buffer's cycle
            if $m.index >$ BUFFER_SIZE - 1 then
                $r.index\_base[i] \leftarrow m.index\_base +$ BUFFER_SIZE

                $r.buffer\_cycle[i] \leftarrow m.buffer\_i\_cycle + 1$          ▷ $i$ is the flow of $pkt$
                $m.buffer\_i\_cycle \leftarrow r.buffer\_cycle[i]$

                $m.index\_base \leftarrow r.index\_base[i]$
                $m.index \leftarrow pkt.seqNo - m.index\_base$
            end if
Phase 2 - Store packet in the respective buffer
            $r.buffer\_i[m.index] \leftarrow pkt.payload$
            $r.buffer\_i\_status[m.index] \leftarrow m.buffer\_i\_cycle$
            $m.buffer\_1\_status \leftarrow r.buffer\_1\_status[m.index]$
            $m.buffer\_2\_status \leftarrow r.buffer\_2\_status[m.index]$
Phase 3 - Perform XOR
            if $m.buffer\_1\_status == m.buffer\_2\_status$ then
                $m.buffer\_1 \leftarrow r.buffer\_1[m.index]$
                $m.buffer\_2 \leftarrow r.buffer\_2[m.index]$
                $pkt.payload \leftarrow m.buffer\_1 \oplus m.buffer\_2$          ▷ $\oplus$ is bitwise XOR

                Multicast($pkt$)
            end if
        end if
Phase 4 - Forwarding rules for normal traffic
    else if $m.drop\_flag ==$ TRUE then
        Drop($pkt$)
    else
        Multicast($pkt$)
    end if
end procedure
```

the P4-XOR switch operation is composed of an initialization sequence and the data plane pipeline itself. We present these algorithms also as an opportunity to have a closer look on the interaction between registers and metadata.

Algorithm 1 corresponds to the initialization sequence. First, it loads the P4 constants. PAYLOAD_SIZE and BUFFER_SIZE are used to specify the size of related structures (i.e., registers, header declarations, and/or header/metadata fields) and are used in logical conditions within the pipeline. Then, when the P4-XOR switch is up and running, it uses runtime CLI commands to populate the match+action tables (line 10). In the same manner, it loads the *nc_enabled_flag* to a stateful and persistent register, and the initial cycle of each buffer (lines 12-15).

As of § 3.1, the Buffering, Coding and Forwarding Modules compose the data plane pipeline and are encapsulated in algorithm 2. It is executed for each arriving *pkt*.

The algorithm starts by loading the *nc_enabled_flag* and *buffer_cycles* to metadata (lines 2-4). We use metadata fields to hold information already present at the registers because primitive actions cannot directly take registers as parameters.

If the *pkt* is valid (i.e., it's from one of the encoding flows), then the respective *index_base* is loaded to metadata and the *index* is calculated. If the *index* is greater than the upper bound, then the *pkt* is discarded (Phase 1). However, if it is within the upper bound but is greater than the size of the buffer, then its *buffer_cycle*, *index_base* and current *index* are updated (Phase 1.1).

Thereafter, given the *index*, the *pkt.payload* is stored at the respective *buffer* and its *buffer_status* is updated with its current *buffer_cycle*. The *buffer_status* from all buffers are also fetched, given the *index*. (Phase 2)

Finally, if the *buffer_status* are all equal, then there is an encoding opportunity. Hence, given the *index*, all *pkt.payloads* are gathered from their respective *buffers*, and the current *pkt.payload* is updated with the bitwise XOR of them all. (Phase 3)

If the *pkt* is invalid (i.e., it's normal traffic) and the *drop_flag* was activated during parsing, then the packet is discarded. Otherwise, it is multicasted on the remaining ports of the switch, by default. (Phase 4)

3.3 P4-RLNC Switch

The P4-RLNC switch deploys a single intraflow RLNC variant scheme. The size of the symbol's elements, and all arithmetic operations are over $GF(2^8)$.

Just as we did for the P4-XOR switch, we propose a protocol header (§ 3.3.1), which fits our needs. We review the design of the parser (§ 3.3.2), buffering module (§ 3.3.3), coding module (§ 3.3.4) and forwarding module (§ 3.3.5). Finally, we consolidate the switch's operation (§ 3.3.6) in algorithmic format.

3.3.1 Protocol Header Format

Below is a detailed low-level representation of the RLNC protocol header.

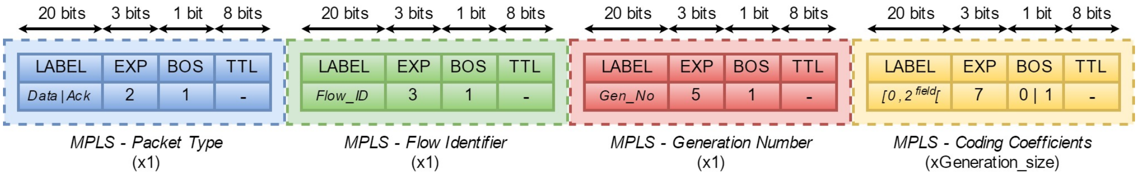


Figure 3.5: P4-RLNC Switch's Protocol Header.

The MPLS header stack is composed by four kinds of headers. As previously mentioned in § 3.2.1, the LABEL field holds the information of the MPLS header per se, the EXP field uniquely identifies the type of header within the protocol, and finally the BOS field pinpoints the last header of each type. The TTL field is not currently used.

The first kind identifies the packet's type[4]. There are two types: DATA and ACK. A DATA type defines a packet that contains an encoded symbol. A ACK type defines a packet that the destination node sends after successfully decoding a whole generation of symbols, so that the source can advance to the next generation.

The second is a flow identifier[5]. Currently, our solution is *intraflow* (only one flow is used and RLNC is performed within that flow), and so, this header is not really that useful. However, if more flows are considered, RLNC can be extended to a multiple *intraflow*, or even an *interflow* coding scheme. In that case, this header is more useful.

The third identifies the generation[6]. If present on a DATA packet, then it represents the generation of the encoded symbol contained. In the case of an ACK packet, it is the generation that the destination has successfully decoded.

The fourth kind of header is exclusively used on a DATA type to hold the coefficients[7] used on the encoding of the symbol. Each coefficient is a value within the GF considered. As the packet is recoded at intermediate nodes, the coefficients are updated.

[4] Referred interchangeably as *pkt.type*, from now on.

[5] Referred interchangeably as *pkt.flowID*, from now on.

[6] Referred interchangeably as *pkt.genID*, from now on.

[7] Referred interchangeably as *pkt.coef*, from now on.

Putting it all together, each packet has one of the following formats:

- pkt = (pkt.type = DATA, pkt.flowID, pkt.genID, pkt.coefs, pkt.payload)
- pkt = (pkt.type = ACK, pkt.flowID, pkt.genID)

Again, we rely on MPLS for the same reasons as for the P4-XOR switch (§ 3.2). The generation size and galois field considered are the only information that must be well established among all the source, intermediate and destination nodes. Obviously, the format of this protocol header must be agreed upon and respected as well.

3.3.2 Parse Graph

The high-level representation of the P4-RLNC switch parser is represented in Figure 3.6.

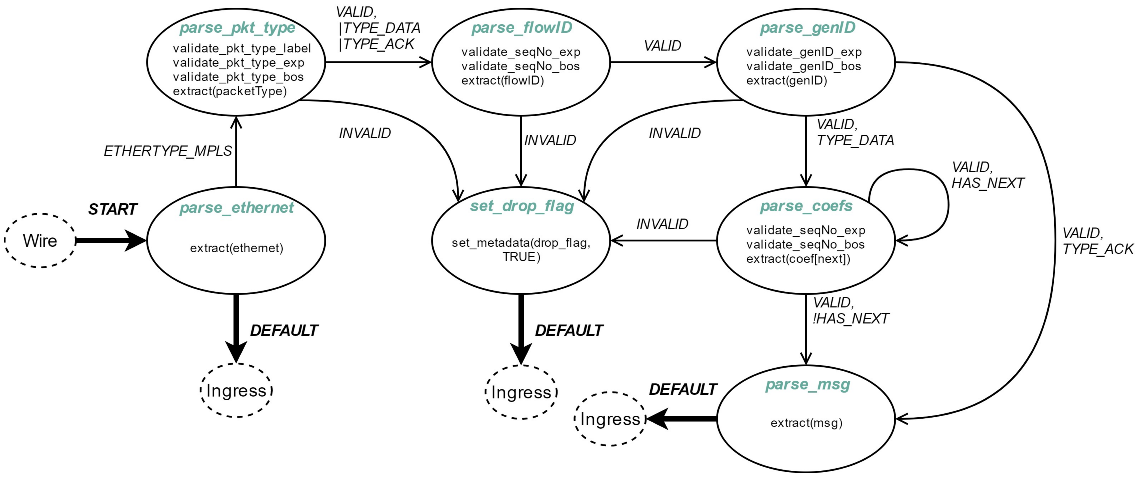


Figure 3.6: High-level representation of the P4-RLNC Switch's parser.

It operates similarly to the P4-XOR switch parser. First, it checks for the presence of an Ethernet header and if it indicates the protocol of the encapsulated frame's is MPLS, then it validates and extracts the MPLS header stack, according to the format previously proposed. If the packet violates this format, it is marked to be later dropped at the ingress control flow. If the MPLS header stack is not present, then the packet is not part of RLNC, and is immediately returned to the control ingress flow to be multicasted.

The only difference is that the remaining payload is not extracted to a single header, but to multiple ones instead (one header for each element of the symbol). This is because in RLNC, the multiplications and additions used during encoding/recoding/decoding are performed over the GF, and therefore over the elements of symbols, which are values within the GF as well. We need header instances to reference each element individually, in order to properly implement the GF arithmetic, characteristic of the recoding process for a RLNC scheme. For instance, if the GF is 2^8 and DATA packets have 32 bits, then their *pkt.payload* represent a symbol with 4 elements (i.e., of 8 bits each) and thus, the *pkt.payload* is extracted to a stack of 4 headers.

3.3.3 Buffering Module

Similarly to the P4-XOR switch (§ 3.2.3), packets from the encoding flow have to be stored, in order to be available for future encoding opportunities. In this case, the encoding traffic consists of packets with *pkt.type* equal to DATA. Internally, the P4-RLNC switch keeps track of the current generation (*gen_current*) and the identifier of the encoding flow (*flowID*). To keep it short, a *pkt* is buffered if its *pkt.type* = DATA, *pkt.genID* = *gen_current* and *pkt.flowID* = *flowID*. Buffering is exclusively done for DATA packets.

Keep in mind that a whole arriving *pkt.payload* is a symbol. Also, GF arithmetic is performed over the symbol's elements. As such, for each element, we have a buffer. (see Fig. 3.7).

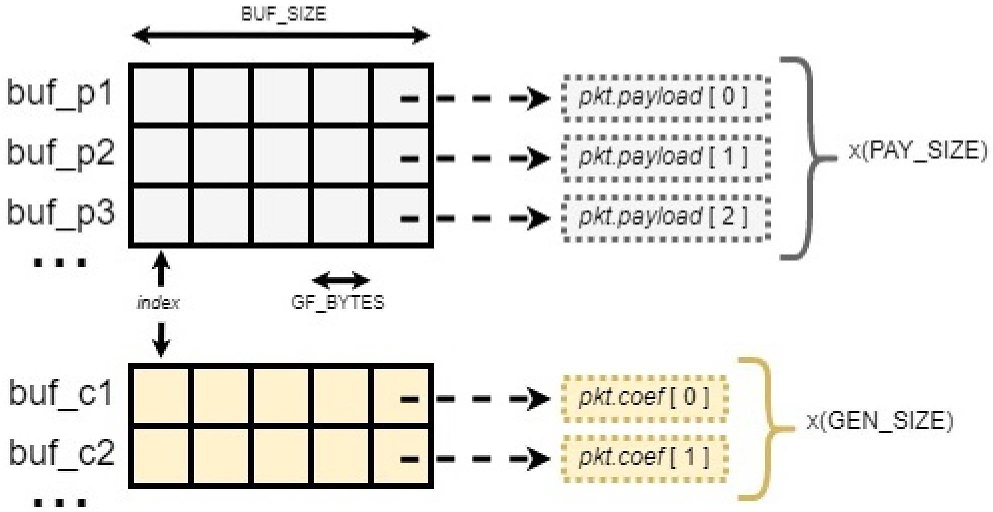


Figure 3.7: P4-RLNC Switch's Buffering Model.

Let's call them payload buffers **(buf_p)**. For instance, if we consider a network where *pkt.payloads* (i.e., symbols) are expected to hold three elements (represented by PAY_SIZE), then we have three distinct payload buffers: *buf_p1*, *buf_p2* and *buf_p3*. The buffer's cells and symbol's elements must share the same bit-width (GF_BYTES). As we consider $GF(2^8)$, then 8 bits are necessary, and sufficient, to hold all GF arithmetic results ranging from 0 to 255. Therefore, GF_BYTES equals 8.

The same applies for the encoding coefficients. A *pkt* will have a number of *pkt.coefs*, equal to the generation size (GEN_SIZE). We also need buffers to store them. Let's call them coefficient buffers **(buf_c)**. For instance, if the GEN_SIZE is two, then each *pkt* has a vector of two *pkt.coefs* and we have two distinct coefficient buffers: *buf_c1* and *buf_c2*.

For indexation, the P4-RLNC switch has an internal *index*. This *index* points to the first free column across all *buf_p* and *buf_c* buffers. Storage is done vertically, meaning that each column holds all the data of one single *pkt*. If *index* is less than the size of the buffers (BUF_SIZE), then there are still free columns. The *pkt.payload* and *pkt.coefs* are stored, respectively, in the payload and coefficient buffer's at the column given by *index*. Afterwards, *index* is updated by one. Otherwise, if *index* is equal than or greater to BUF_SIZE, then the buffers are full. In that case, the *pkt* overwrites an occupied column, chosen randomly by an *index_r*. From the moment when the buffers are full, *index_r* is computed and used instead, until we have to step to the next generation.

The P4-RLNC switch knows that it has to update the current generation when it receives an ACK packet. If the *pkt.type* = ACK, *pkt.genID* = *gen_current* and *pkt.flowID* = *flowID*, then it increments *gen_current* by one, and resets the buffers, by changing the *index* to zero.

As we will see in § 3.3.4, the *index* controls which packets are retrieved from the buffer for the recoding process. As such, resetting the *index* is the same as resetting the buffers. There is no need to delete the contents of each buffer's cell.

Just like in the case of the P4-XOR Switch, the Buffering and Coding Modules are applied to encoding traffic, if the RLNC functionality of the P4-RLNC Switch is activated (dictated by the *nc_enabled_flag* register).

3.3.4 Coding Module

The Coding Module for the P4-RLNC switch is a variant of Standard RLNC (§ 2.1.4). As explained, the solution is intraflow as we mix together packets within a single flow. Figure 3.8 summarizes our model.

As a reminder, and building from § 2.1.4, we consider a network composed of a single source node, multiple intermediate nodes and a single destination node. GEN_SIZE represents the size of each generation, PAY_SIZE the number of elements of a symbol, and GF_BYTES the bit-width of each element within a symbol. Among all nodes, these values are assumed to be well known.

The starting point is the source, which has a message to send to the destination node. First, it breaks the message in generations. The generations are further divided into GEN_SIZE original symbols with PAY_SIZE elements of GF_BYTES of size each (see Fig. 3.8 - top left). Given the current generation, the source creates and sends DATA packets, until an ACK packet for this generation is received. The DATA packet is created by

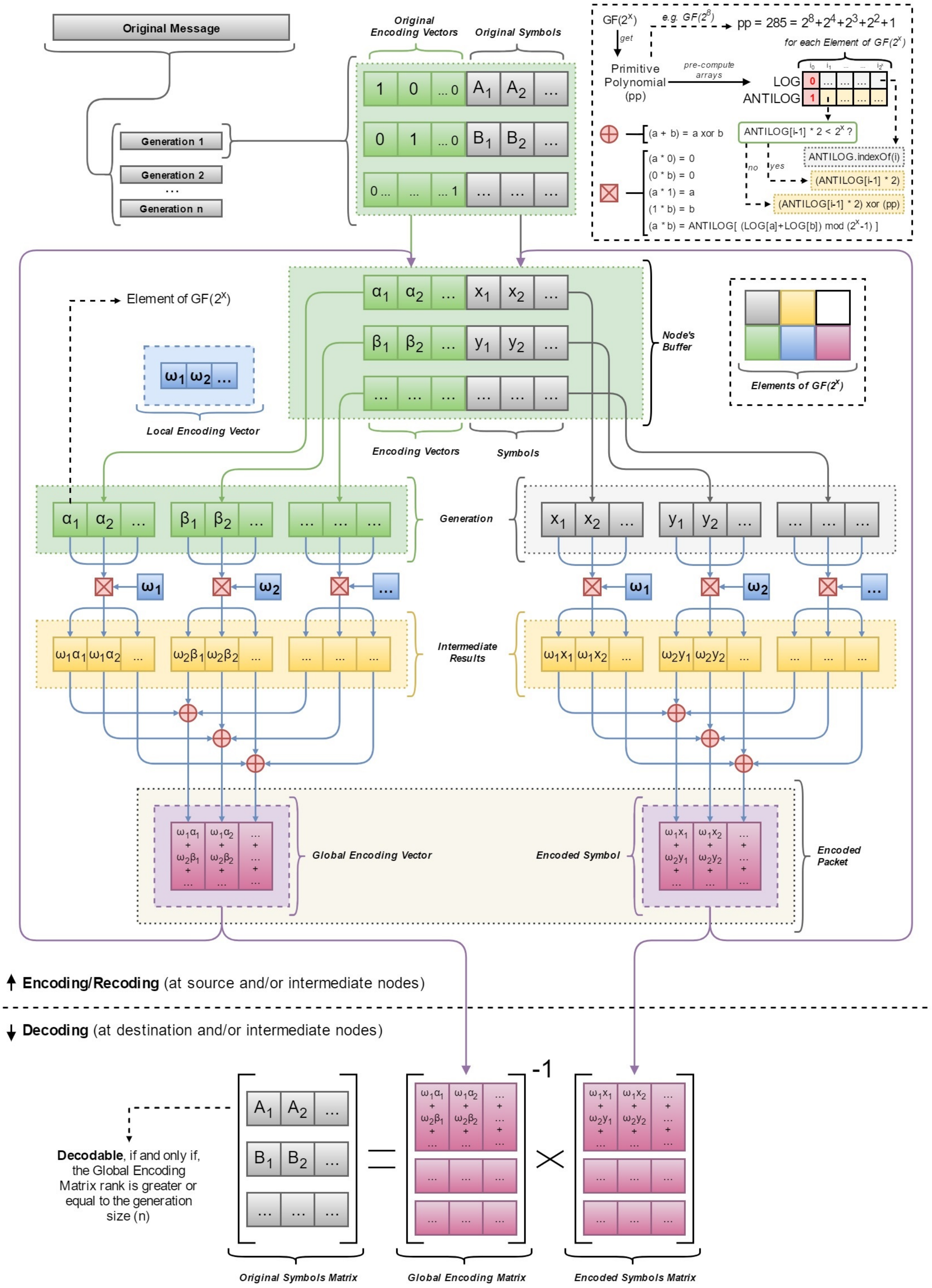


Figure 3.8: P4-RLNC Switch's Coding Model - Complete Overview.

randomly obtaining a local encoding vector of GEN_SIZE coefficients and applying the encoding/recoding process (see Fig. 3.8 - center).

The destination node organizes a number of GEN_SIZE *pkt.payloads* and *pkt.coefs* from DATA packets in two matrices, until it is able to decode the current generation and send an ACK packet (see Fig. 3.8 - bottom).

Intermediate nodes represent P4-RLNC switches and are where recoding takes place. Our recoding scheme is a variant of the Standard RLNC, which we refer to as **Sparse RLNC with Incremental Density**. The *density* is related with the number of packets used as input to the encoding/recoding process. For instance, a recoding density of three implies that three packets are used as input. This means that the number of symbols (i.e., *pkt.payloads*) and respective coefficient vectors (i.e., *pkt.coefs*) from buffered packets, used as input to the encoding/recoding process, is not fixed at GEN_SIZE (the required number for **Standard RLNC**), but is influenced by the number of buffered packets (given by *index*).

To be more specific, after being subject to the Buffering Module, a DATA packet is ready to be recoded (see Fig. 3.8 - center). If the *index* is greater than or equal to GEN_SIZE, then we have enough packets for **Standard RLNC**[8]. In that case, GEN_SIZE buffered packets are used as input. The packets are chosen randomly, without repetition (more details later on). For each chosen packet, we retrieve its PAY_SIZE symbol's elements and GEN_SIZE coefficients, from the payload buffers and coefficient buffers, respectively.

Else, if *index* is less than GEN_SIZE, then we perform **Sparse RLNC with Incremental Density**. In other words, we use all the available packets from the buffers. For instance, if *index* equals three, then three packets are used. On an important note, BUF_SIZE should be greater than or equal to the GEN_SIZE, to perform Standard RLNC eventually.

About the arithmetic operations, of addition and multiplication, involved in RLNC. Bare in mind that we are using a protocol header with an specific format and restricted in size. Additionally, due to P4-14 constrictions on the size of the declared data structures, buffers are restricted in size as well. As a consequence conventional arithmetic cannot be used, as these structures will eventually overflow (for obvious reasons). Therefore, we need to use GF arithmetic, which keeps all results within a certain range of values. One way to implement a given $GF(2^x)$ is to compute a primitive polynomial (pp), which is used calculate a set of log and antilog values. These values are used to define GF arithmetic operations (Fig. 3.8 - top right). We provide a more detailed description below.

Galois Field Arithmetic

Because we are restricted in size, if conventional arithmetic was used, large calculations could result in overflows. To deal with this problem, all the arithmetic additions and multiplications, involved in the encoding/recoding/decoding processes, are performed over a

[8] At this point the code is *sparse*, as not all buffered packets are used for recoding

given galois field (GF). This way, all inputs and outputs of additions and multiplications share the same bitwise size.

Remind that we opted for a $GF(2^8)$, meaning that all chosen coefficients, symbol's elements and, consequently all addition's and multiplication's inputs and outputs have 8 bits (i.e., 1 byte).

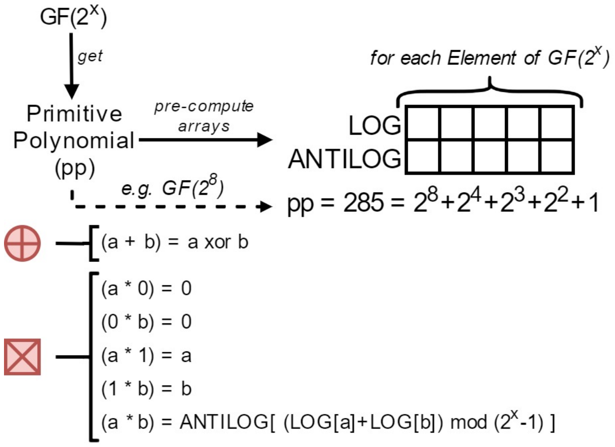

GF(2^x)
get
Primitive Polynomial (pp)
pre-compute arrays
for each Element of GF(2^x)
LOG
ANTILOG
e.g. GF(2^8)
pp = 285 = 2^8+2^4+2^3+2^2+1
(a + b) = a xor b
(a * 0) = 0
(0 * b) = 0
(a * 1) = a
(1 * b) = b
(a * b) = ANTILOG[(LOG[a]+LOG[b]) mod (2^x-1)]

Figure 3.9: Arithmetic addition and multiplication over a galois field.

The implementation of addition is simple. To add a pair of values, an XOR is the only requirement. Multiplication is far more complex. First, given the GF considered, a primitive polynomial (pp) is calculated, which is then used to pre-compute a LOG and ANTILOG lookup table, for each element of the field. To multiply a pair of values, these tables are indexed (see Fig. 3.9 and 3.10).

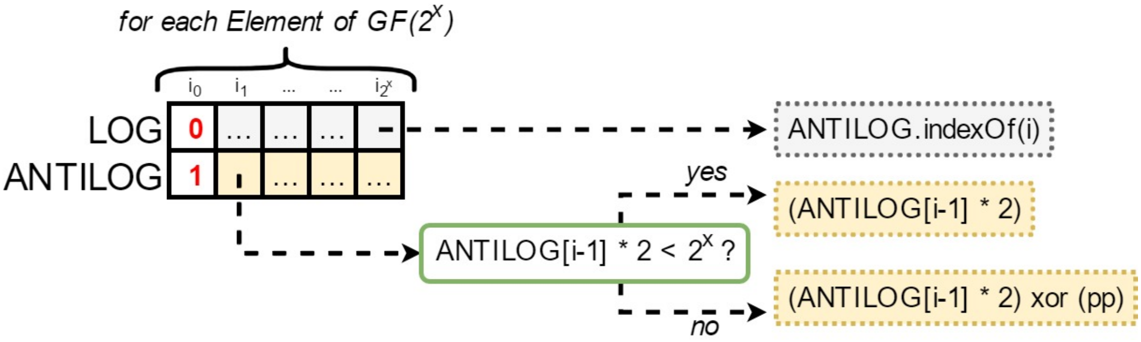


Figure 3.10: LOG and ANTILOG table pre-computation.

To speed up the processing of each packet, the P4-RLNC switch saves these pre-computed tables in stateful memory (i.e., registers), during startup. It is also important to note that if-statements could in principle be required to handle the 0 and 1 multiplication cases, which can slow down processing by a considerable amount. However, we remove the need for the if-statements by extending the ANTILOG table [48] to handle these cases.

For this purpose, the ANTILOG table becomes four times larger. The idea is to repeat the set of values on more time and fills the rest with 0. The LOG table changes to 512 at index 0. This way, if either inputs of multiplication is zero, the offset at the ANTILOG table is in the zone filled with zeros, and so the result is 0, as it should be. The multiplication by one is handled in the same manner. Because index 1 of the LOG table has value 255, it places the offset at the ANTILOG table on the second set of values, for the second input. Note that this is the same as performing no multiplication at all, as it should be. (see Fig. 3.11)

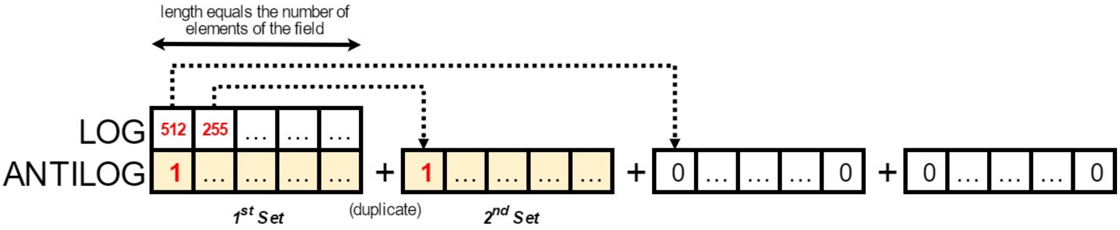
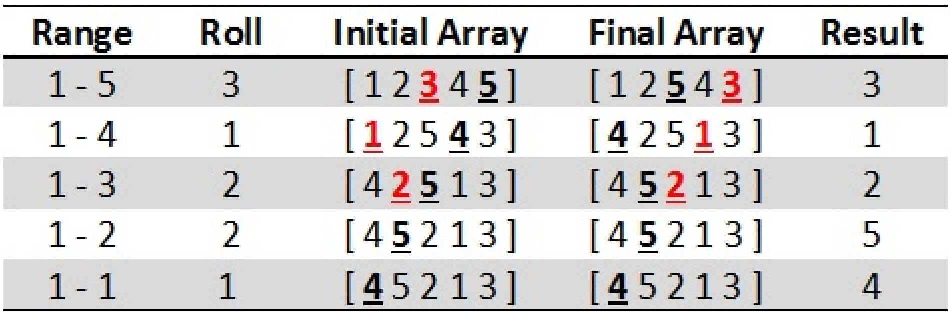


Figure 3.11: LOG and ANTILOG table pre-computation - An improvement.

So, at a slightly increase in memory used, the overall performance of the switch is greatly improved.

Choosing Random Packets

As already mentioned (§ 3.3.4), when the number of stored packets exceeds the generation size, a number of packets, equal to the generation size, are randomly fetched from the buffers, each time recoding is performed.



Range	Roll	Initial Array	Final Array	Result
1 - 5	3	[1 2 **3** 4 **5**]	[1 2 **5** 4 **3**]	3
1 - 4	1	[**1** 2 5 **4** 3]	[**4** 2 5 **1** 3]	1
1 - 3	2	[4 **2** **5** 1 3]	[4 **5** **2** 1 3]	2
1 - 2	2	[4 **5** 2 1 3]	[4 **5** 2 1 3]	5
1 - 1	1	[**4** 5 2 1 3]	[**4** 5 2 1 3]	4

Figure 3.12: Modern variation of Fisher-Yates shuffle algorithm - An example.

In order to randomly choose packets without repetition, we implement Durstenfeld's modern variant of the Fisher-Yates shuffle algorithm [52]. The algorithm works as follows:

1. Fill an array with the numbers to be randomly picked and set a *last_index* to the array length.

2. Roll a number n between [1, *last_index*].

3. Swap array[n] with array[*last_index*].

4. Return the new number at array[*last_index*].

5. Update *last_index* = *last_index* - 1.

6. Repeat steps 2-5, while *last_index* > 0.

Figure 3.12, illustrates the algorithm in practice. The choice of this algorithm was due to its simplicity and for fitting well within the constraints of network switches. The array just needs to be filled with the all the buffers indexes (i.e., from 0 to BUF_SIZE - 1), with *last_index* indicating the last occupied column (i.e., given by *index* - 1).

3.3.5 Forwarding Module

The Forwarding Module follows the same methodology as that of the P4-XOR Switch (§ 3.2.3). All traffic is multicasted on the remaining ports of the switch, by default. However, the encoding traffic is dropped, if the proposed MPLS header stack format is violated.

3.3.6 Operation

Putting it all together, we present in algorithmic form the operation of the P4-RLNC Switch. Again, we detail these algorithms to have a closer look at the necessary registers and metadata, and how they are related.

Algorithm 3 Initialization of the P4-RLNC Switch

```
1: procedure INITIALIZE_RLNC_SWITCH()
2:     // Load global and parser constants
3:     ...
4:
5:     Populate_Tables("lnc_cmd.txt")
6:     Load_Registers("lnc_cmd.txt")                ▷ In this case, $r.nc\_enabled\_flag$
7:     Load_Log_Table("GF256_log_cmd.txt")
8:     Load_InvLog_Table("GF256_invlog_cmd.txt")
9: end procedure
```

Algorithm 3 presents the initialization sequence of the switch. It starts by loading the constants (line 2). Thereafter, via its runtime CLI interface, the match+action tables are populated (line 3) and the *nc_enabled_flag* is set to a register (line 4). Finally, it loads the LOG and ANTILOG pre-computed tables to arrays of registers.

Algorithm 4 Data Plane of the P4-RLNC Switch

```
 1: procedure RLNC_DATA_PLANE()
 2:     if $m.drop\_flag$ == TRUE then
 3:         DROP($pkt$)
 4:     else
 5:         MULTICAST($pkt$)
 6:     end if
 7:     $m.nc\_enabled\_flag \leftarrow nc\_enabled\_flag$
 8:     $m.gen\_current\_flag \leftarrow gen\_current\_flag$
Phase 1 - Set/Get current generation
 9:     if $m.nc\_enabled\_flag$ == TRUE then
10:         if $m.gen\_current\_flag$ == FALSE then
11:             $r.gen\_current \leftarrow pkt.genID$
12:             $r.gen\_current\_flag \leftarrow$ TRUE
13:         end if
14:         $m.gen\_current \leftarrow r.gen\_current$
15:         $m.gen\_current\_flag \leftarrow r.gen\_current\_flag$
```

Algorithm 4 consolidates the Buffering, Coding and Forwarding Modules. This algorithm is fully applied to each arriving packet. First, it applies the forwarding rules. All traffic (normal and encoding) is set for multicast, by default (line 5). Encoding traffic that violates the proposed protocol header format, had the *drop_flag* activated during parsing, and as such, is immediately dropped (lines 2-3).

Then, the *nc_enabled_flag* and *gen_current_flag* are loaded to metadata (lines 7-8). If enabled for with RLNC functionality (line 9), the remaining of the algorithm is applied for encoding traffic. Normal traffic is not subject to further processing in this pipeline, as it is neither a DATA or ACK packet. Right after startup, the P4-RLNC Switch cannot simply guess the current generation. As such, it assumes the current generation as the one of the first arriving packet from encoding traffic. The *gen_current_flag* indicates if the first generation identifier was set or not. If it wasn't, the switch sets it to the one contained on the packet (lines 11-12). Otherwise it just loads to metadata the one it has stored in a register (lines 14-15). Afterwards, we check if it is a DATA or an ACK packet.

In the case of a DATA packet, its coefficient vector must be stored on the *coefficient buffers* (buf_c) and its payload on the *payload buffers* (buf_p). The first step is to get the *index* (buf_index) of the first free column. If the index is within the buffer size (BUF_SIZE), then the contents of the packet are stored, in the respective buffers, at that position and the index is updated (Phase 2.1).

Otherwise, if the buffers are full, then a random index (buf_index_r) is chosen and the respective column across all the buffers is overwritten (Phase 2.2).

Finally, the packet is recoded. The number of packets retrieved and used on the recoding

Phase 2.1 - Store packet at next free index

```
        $m.buf\_index \leftarrow r.buf\_index$
        **if** $pkt$ **is DATA then**
            **if** $m.buf\_index <$ BUF_SIZE **then**
                $m.buf\_index \leftarrow r.buf\_index$
                $r.buf\_c1[m.buf\_index] \leftarrow pkt.coef[0]$
                $r.buf\_c2[m.buf\_index] \leftarrow pkt.coef[1]$
                ... (repeated ×GEN_SIZE times)

                $r.buf\_p1[m.buf\_index] \leftarrow pkt.payload[0]$
                $r.buf\_p2[m.buf\_index] \leftarrow pkt.payload[1]$
                ... (repeated ×PAY_SIZE times)

                $r.buf\_index \leftarrow m.buf\_index + 1$
                $m.buf\_index \leftarrow r.buf\_index$
```

Phase 2.2 - Store packet at random index

```
            **else**
                $m.buf\_index\_r \leftarrow rng(0,$ BUF_SIZE$)$
                $r.buf\_c1[m.buf\_index\_r] \leftarrow pkt.coef[0]$
                $r.buf\_c2[m.buf\_index\_r] \leftarrow pkt.coef[1]$
                ... (repeated ×GEN_SIZE times)

                $r.buf\_p1[m.buf\_index\_r] \leftarrow pkt.payload[0]$
                $r.buf\_p2[m.buf\_index\_r] \leftarrow pkt.payload[1]$
                ... (repeated ×PAY_SIZE times)
            **end if**
```

Phase 3 - Perform RLNC

```
            **if** $m.buf\_index >$ GEN_SIZE **and** $m.buf\_index \leq$ BUF_SIZE **then**
                **RECODE**(R)
            **else if** $m.buf\_index == 1$ **then**
                **RECODE**(1)
            **else if** $m.buf\_index == 2$ **then**
                **RECODE**(2)
            **end if**
            ... (repeated ×GEN_SIZE times)
```

Phase 4 - Update current generation

```
        **else if** $pkt$ **is ACK then**
            $r.gen\_current \leftarrow m.gen\_current + 1$
            $r.gen\_current\_flag \leftarrow$ TRUE
            $r.buf\_index \leftarrow 0$
        **end if**
    **end if**
**end procedure**
```

process depends on the generation size and number of packets stored. While the buffers have a number of packets that is less than or equal to the generation size, all stored packets are used for recoding. When the number of stored packets exceeds the generation size, a number of packets, equal to the generation size, is randomly retrieved from the respective buffers (without repetition) and used for recoding (Phase 3).

In the case of an ACK packet, the current generation that the switch accepts and recode is incremented and the buffers are reset (by changing the index of the first free cell to 0). This results in packets from older generations, that might arrive at the switch, to be set for multicast and ignored by the Buffering and Coding Modules (Phase 4).

Algorithm 5 The RLNC recoding procedure

```
1: function RECODE(n)
2:     Compute random coefficient vector to $m$
3:     Load $n$ packets to $m$        ▷ If $n == R$, randomly chosen with Fisher-Yates algorithm
4:     Update $pkt$ payload
5:     Update $pkt$ coefficients
6: end function
```

The recoding procedure is depicted in algorithm 5. First, we randomly obtain GEN_SIZE coefficients within the GF, which are loaded to metadata (line 1). Then, for each packet to be retrieved for recoding, we get its symbol's elements from the payload buffers, and its coefficient vector from the coefficient buffers, to metadata as well (line 2). Then, using all this as input, the current packet is recoded (lines 4-5).

3.4 Summary

This chapter provided an overview over the design of our switches (§ 3.2, § 3.3). We began by proposing an overlying architecture (§ 3.1), which organizes the functionality in several modules. Following that, for each of our switches, we proposed a protocol header stack (§ 3.2.1, § 3.3.1) that packets must follow; described the parsers (§ 3.2.2, § 3.3.2); and the buffering (§ 3.2.3, § 3.3.3), coding (§ 3.2.4, § 3.3.4), and forwarding (§ 3.2.5, § 3.3.5) modules. Finally, we consolidated the whole operation of the processing pipelines in algorithmic format (§ 3.2.6, § 3.3.6).

Chapter 4 – Implementation

At this point, it is already acknowledged that any P4 program consists on the definition of: headers, parsers, tables, actions and control flow functions. In this chapter, we provide a detailed walkthrough over the implementation of all these components, in the context of our solutions: P4-XOR Switch and P4-RLNC Switch.

We begin by defining the headers (§ 4.1) and the parser state machines (§ 4.2). Then, for each switch, we provide the overview over the respective data plane pipelines (§ 4.3, § 4.4), followed by a detailed explanation of their Buffering (§ 4.3.1, § 4.4.1), Coding (§ 4.3.2, § 4.4.2) and Forwarding (§ 4.3.3, § 4.4.3) Modules.

As it will be shown, some parts of the implementation were relatively trivial. Others, however, due to several constraints imposed by the language (many of which are themselves imposed by the switching hardware), were very challenging.

To name just a few, P4 does not support variable bit-widths for data structures (namely, registers and metadata fields). It constraints the bit-width to the declared values, which cannot be changed at runtime. Also, there are no means to create additional data structures, beyond the declared ones, in runtime. At times, this results on having an extensive and complex data structure, as all data structures that one anticipates to be required must be declared explicitly. As the explicitly declared data structures are intrinsic to the correct operation of a P4 program running on a P4-compliant device, changes of network parameters (e.g., GEN_SIZE, in our case), forces turning off the device, modifying the code, recompiling the program, and finally running it again.

Other important constraint is the nonexistence of a built-in operator for creating loops (mainly needed for the complex RLNC recoding process), which forces us to manually implement our owns.

4.1 Header Definitions

We begin with the definition of the headers, which are used to validate and extract the required information (for later use) of arriving packets, during parsing. Listings 4.1 and 4.2 showcase the needed headers, for each of the switches.

Listing 4.1: Headers (P4-XOR)

```
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}
header_type mpls_t {
  fields {
    label : 20;
    exp : 3;
    bos : 1;
    ttl : 8;
  }
}
header_type msg_t {
  fields {
    content: PAYLOAD_SIZE;
  }
}
```

Listing 4.2: Headers (P4-RLNC)

```
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}
header_type mpls_t {
  fields {
    label : 20;
    exp : 3;
    bos : 1;
    ttl : 8;
  }
}
header_type msg_t {
  fields {
    content: GF_BYTES;
  }
}
```

As expected, this step is identical for both cases. We define an MPLS header **(mpls_t)**, because the proposed protocol header stacks (§ 3.2.1 , § 3.3.1) include MPLS. As we want our packets to transverse a conventional LAN, an Ethernet header is also included. As such, we define it as well **(ethernet_t)**. For buffering and encoding, the message's contents contained in each packet (i.e., the payload) also needs to be extracted. Therefore, we define a header for that purpose **(msg_t)**. It is important to justify why we are treating the entire payload as a header. This is due to one limitation of the P4 model: the packet payload is supposed to be sent from *ingress* directly to *egress* without modification. This limitation does not permit computations over the payload, which would preclude a scheme such as NC. The workaround was to consider the payload as another header.

Notice that the only difference between the two solutions is the field size of *msg_t*. The P4-XOR switch will extract the whole payload to a single header, hence *msg_t.content* field size equals the packet's payload size **(PAYLOAD_SIZE)**.

In the case of the P4-RLNC switch, recall that the packet's payload is one symbol and we need individual reference to each of the symbol's elements. Therefore, the *msg_t.content* field size equals the size of an element **(GF_BYTES)**, and the payload/symbol will be a stack of *msg_t* headers.

4.2 Parser State Machines

For the creation of our parser's state machine, and accordingly to the protocol header stacks, first we define some constants (Listings 4.3 and 4.4). The constants FLOW_ID, FLOW_ID_1, FLOW_ID_2, TYPE_DATA and TYPE_ACK might not necessarily be what is shown, and thus be set to other values, as long as all the nodes use and respect the same values. Then, we instantiate headers from the previous header declarations (Listings 4.5 and 4.6), to which the packet's contents are to be extracted. Notice the required header stacks for both the coefficients (mpls_t coef) and symbol's element within the payload (mpls_t msg), for the P4-RLNC case.

Listing 4.3: Parser Constants (P4-XOR)

```
#define  ETHERTYPE_MPLS  0x8847

#define  MPLS_EXP_FLOW_ID  3
#define  MPLS_EXP_SEQ_NO   4



#define  FLOW_ID_1  500
#define  FLOW_ID_2  501
```

Listing 4.4: Parser Constants (P4-RLNC)

```
#define  ETHERTYPE_MPLS  0x8847

#define  MPLS_EXP_PKT_TYPE  2
#define  MPLS_EXP_FLOW_ID   3
#define  MPLS_EXP_GEN_ID    5
#define  MPLS_EXP_COEF      7

#define  FLOW_ID  13579

#define  TYPE_DATA  1234
#define  TYPE_ACK   5678
```

Listing 4.5: Header Instances (P4-XOR)

```
header  ethernet_t  ethernet;

header  mpls_t  mpls_flowID;
header  mpls_t  mpls_seqNo;



header  msg_t  msg;
```

Listing 4.6: Header Instances (P4-RLNC)

```
header  ethernet_t  ethernet;

header  mpls_t  packetType;
header  mpls_t  flowID;
header  mpls_t  genID;
header  mpls_t  coef[GEN_SIZE];

header  msg_t  msg[PAY_SIZE];
```

Finally, the parser's state machines are created. We provide an excerpt of some states within our parsers. Listing 4.7 exemplifies the starting state, and the validation/extraction of the ethernet header. If the packet contains an ethernet header with MPLS as etherType, then it might be encoding traffic and parsing continues. Listing 4.8 provides the full validation and extraction of an MPLS header for the protocol header stack. This procedure is identical for the remaining headers.

Listing 4.7: Starting and Ethernet (P4-XOR)

```
parser start {
  return parse_ethernet;
}

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    ETHERTYPE_MPLS : validate_flowID_label;
                default : ingress;
  }
}
```

Listing 4.8: Flow Identifier (P4-XOR)

```
parser validate_flowID_label {
  return select(current(0,20)) {
    FLOW_ID_1 : validate_flowID_exp;
    FLOW_ID_2 : validate_flowID_exp;
          default : set_drop_flag;
  }
}
parser validate_flowID_exp {
  return select(current(20,3)) {
    MPLS_EXP_FLOW_ID : validate_flowID_bos;
                  default : set_drop_flag;
  }
}
parser validate_flowID_bos {
  return select(current(23,1)) {
          1 : extract_flowID;
    default : set_drop_flag;
  }
}

parser extract_flowID {
  extract(mpls_flowID);
  return validate_seqNo_exp;
}
```

Listing 4.9: Coefficients (P4-RLNC)

```
parser validate_lnc_coef {
  return select(current(20,3)) {
    MPLS_EXP_COEF : extract_lnc_coef;
              default : set_drop_flag;
  }
}

parser extract_lnc_coef {
  extract(coef[next]);
  return select(latest.bos) {
          1 : parse_msg;
    default : validate_lnc_coef;
  }
}
```

Listing 4.10: Payload (P4-RLNC)

```
parser parse_msg {
  extract(msg[next]);
  return parse_msg;
}
```

Listing 4.11: Payload (P4-XOR)

```
parser parse_msg {
  extract(msg);
  return ingress;
}
```

Listing 4.12: Invalidation (Both)

```
parser set_drop_flag {
  set_metadata(p_meta.drop_flag, TRUE);
  return ingress;
}
```

Recall that currently, bmv2 does not support packet discarding at the parsing stage. Invalid packets (i.e., violating the protocol) are marked as invalid, to later be dropped at the data plane pipeline, by triggering a flag (Listing 4.12).

The validation and extraction of coefficients is somewhat different. As its number might be considerably high, we create a recursive loop (Listing 4.9). The same applies for the extraction of the symbol's elements within the packet's payload (Listing 4.10). The difference here is that an explicit return to *ingress* or other different state, is not required as these are the last packet's bytes and return to *ingress* is automatic. For the P4-XOR switch, we extract it to a single header (Listing 4.11).

4.3 P4-XOR Switch

Before going to implementation details of the core functionality of the P4-XOR Switch (i.e., how the buffering, coding and forwarding modules are mapped to metadata, registers, tables and actions), we opted for providing the P4 equivalent of our data plane pipeline first (Listing 4.13). By taking a closer look, it is almost (in practice) a direct mapping of algorithm 2 to table appliance form.

Listing 4.13: Data Plane Pipeline (P4-XOR)

```
control ingress {

  apply(table_load_flags);
  if(meta.nc_enabled_flag == TRUE) {

    apply(table_get_buffer_indexes_1);
    if(meta.index > ((2∗BUFFER_SIZE) − 1)) {
      apply(table_drop);
    }
    else {
      if(meta.index > (BUFFER_SIZE − 1)) {
        apply(table_update_index_base);
        apply(table_update_buffer_cycle);
        apply(table_get_buffer_indexes_2);
      }
      apply(table_write);
      if(meta.buffer_1_status == meta.buffer_2_status) {
        apply(table_XOR);
        apply(table_multicast);
      }
    }
  }
  else {
    if(parser_meta.drop_flag == TRUE) {
      apply(table_drop);
    }
    else {
      apply(table_multicast);
    }
  }
}

control egress {
  // declared and left empty on purpose
}
```

The *ingress* control flow function fully implements the data plane pipeline. It offers an overview over all the existing tables and its application sequence. In the following subsections (§ 4.3.1, § 4.3.2, § 4.3.3), we refer to some of these tables to detail the most relevant parts of the implementation (e.g., **table_write**, **table_XOR** and **table_multicast**) and the faced challenges. The remaining tables, despite basic arithmetic in some cases (e.g., **table_update_buffer_cycle**), end up being simple transitions of data between metadata and registers, suggested by the table names. Therefore, the code is not displayed.

4.3.1 Buffering Module

Deciding on the required data structures was not challenging and came directly from the design phase (§ 3.2.3).

Listing 4.14: Global Constants (P4-XOR)

```
#define  TRUE    1
#define  FALSE   0

#define  PAYLOAD_SIZE   4096
#define  BUFFER_SIZE     1000
```

Listing 4.15: Metadata (P4-XOR)

```
header_type meta_t {
  fields {
    nc_enabled_flag : 1;

    index : 32;
    index_base : 32;

    buffer_1 : PAYLOAD_SIZE;
    buffer_2 : PAYLOAD_SIZE;

    buffer_1_cycle : 8;
    buffer_2_cycle : 8;

    buffer_1_status : 8;
    buffer_2_status : 8;
  }
}
metadata meta_t meta;
```

Listing 4.16: Registers (P4-XOR)

```
register nc_enabled {
  width: 1;
  instance_count: 1; }

register index_base {
  width: 32;
  instance_count: 2; }

register buffer_1 {
  width: PAYLOAD_SIZE;
  instance_count: BUFFER_SIZE; }
register buffer_2 {
  width: PAYLOAD_SIZE;
  instance_count: BUFFER_SIZE; }

register buffer_1_status {
  width: 8;
  instance_count: BUFFER_SIZE; }
register buffer_2_status {
  width: 8;
  instance_count: BUFFER_SIZE; }

register buffer_cycle {
  width: 8;
  instance_count: 2; }
```

We define some constants (Listing 4.14), which are used for the declaration of metadata (Listing 4.15) and registers (Listing 4.16). Again, recall that registers cannot be directly used as a parameter within logical conditions (i.e., if-else statements at *ingress*) and/or other compound actions (e.g., bit_xor, modify_field). That's why a metadata header instance is needed. Its fields are used to hold the required information, for parameterization.

When the pipeline reaches the phase to store a *pkt.payload*, the **table_write** (Listing 4.17) is applied. The table has the *pkt.flowID* as a match because there is an encoding action, per flow. Listing 4.18 demonstrates the whole process of one of these actions: from loading the *pkt.payload* to its buffer, to updating the *buffer_status* in registers and metadata.

Listing 4.17: The Table

```
table table_write {
  reads {
    mpls_flowID.label : exact;
  }
  actions {
    _nop;
    action_write_1;
    action_write_2;
  }
  size : TABLE_WRITE_SIZE;
}

action _nop() { }
```

Listing 4.18: The Compound Action

```
action action_write_1 () {
  register_write(buffer_1, meta.index, msg.content);
  action_set_buffer_status_1();
  action_get_buffer_status();
}

action action_set_buffer_status_1() {
  register_write(buffer_1_status, meta.index, meta.buffer_1_cycle);
}

action action_get_buffer_status() {
  register_read(meta.buffer_1_status, buffer_1_status, meta.index);
  register_read(meta.buffer_2_status, buffer_2_status, meta.index);
}
```

Listing 4.19: The Table Population

```
table_set_default   table_write   _nop
table_add           table_write   action_write_1   500  =>
table_add           table_write   action_write_2   501  =>
```

One action per flow **(action_write_1, action_write_2)** is required because this table is directly called within the pipeline, which implies that it is populated with the runtime CLI interface beforehand (Listing 4.19). The runtime CLI interface does not support taking a string identifying a register or header/metadata field, to specify the parameters of actions. It only supports integers. Therefore, we have to explicitly declare the register name, to be read from and/or written to, within the action. That's why an action per flow/buffer is required.

As the table's match is the *pkt.flowID*, the table default is set to an action that does nothing **(_nop)**, in the case when the *pkt.flowID* is none of the two we defined.

4.3.2 Coding Module

The encoding operation consists of a single table (Listing 4.20). The procedure is the same for every packet and at this stage all validations are made, hence the table is configured with a single default action (Listing 4.21). This action is as simple as loading to metadata the *pkt.payloads*, using the already computed *index* from the respective buffers; applying the XOR pre-built action; and update the *pkt.payload* of the packet being processed with the result (Listing 4.22).

Listing 4.20: The Table

```
table table_XOR {
  actions {
    action_XOR;
  }
}
```

Listing 4.21: The Table Population

```
table_set_default   table_XOR   action_XOR
```

Listing 4.22: The Compound Action

```
action action_XOR () {
   // Load Payloads to Metadata
   register_read(meta.buffer_1, buffer_1, meta.index);
   register_read(meta.buffer_2, buffer_2, meta.index);

   // Perform encoding: XOR
   bit_xor(meta.buffer_1, meta.buffer_1, meta.buffer_2);

   // Update pkt.payload
   modify_field(msg.content, meta.buffer_1);
}
```

4.3.3 Forwarding Module

As mentioned in the design phase (§ 3.2.5), traffic is multicast by default. To implement this behavior, we have a single table (Listing 4.23), which matches on the *ingress_port* of the current packet. The port is given by the *standard_metadata* of the target switch (bmv2). The multicast action itself (Listing 4.24) consists on setting a multicast group (given by *mcast_grp*). This information is contained within *intrinsic_metadata*, which is target specific and, although not shown, must be declared somewhere in the P4 program.

Listing 4.23: The Table

```
table table_multicast {
   reads {
      standard_metadata.ingress_port : exact;
   }
   actions {
      _drop;
      action_multicast;
   }
}

action _drop() {
   drop();
}
```

Listing 4.24: The Compound Action

```
action action_multicast(group) {
   modify_field(intrinsic_metadata.mcast_grp, group);
}
```

Listing 4.25: The Table Population

```
mc_mgrp_create 1
mc_node_create 0 2
mc_node_create 1 3
mc_node_associate 1 0
mc_node_associate 1 1
table_set_default   action_multicast   _drop
table_add  action_multicast  action_multicast  1 => 1
```

A table population example is presented in Listing 4.25. The first line creates a multicast group with identifier 1. The second and third lines, fetch logical ports 2 and 3. The fourth and fifth lines associate the fetched ports to the multicast group. Finally, we create a match for *ingress_port* 1 to *mcast_grp* 1, and drop incoming packets from the remaining ports by default (**_drop**).

Recall from design (§ 3.2.5) that we change the forwarding of encoding traffic from multicast (on all the remaining ports) to unicast (on a single port) for a subset of tests, during the Evaluation Phase. We provide an example of how to implement an unicast behavior in § 4.4.3.

4.4 P4-RLNC Switch

Listing 4.26 provides the whole picture of the P4-RLNC Switch data plane pipeline, which, in this case, is the conversion of algorithm 4 to the P4 format.

Listing 4.26: Data Plane Pipeline (P4-RLNC)

```
control ingress {

  if(parser_meta.drop_flag == TRUE) {
    apply(table_drop);
  }
  else {
    apply(table_forward);
  }

  apply(table_load_1);
  if(meta.nc_enabled_flag == TRUE) {

    if(meta.gen_current_flag == FALSE) {
      apply(table_set_gen_current);
    }
    apply(table_load_2);

    if(packetType.label == TYPE_DATA and flowID.label == FLOW_ID and genID.label == meta.gen_current){

      if(meta.buf_index < BUF_SIZE) {
        apply(table_write);
        apply(table_update_buffer_index);
      }
      else {
        apply(table_overwrite);
      }
      if(meta.buf_index >  GEN_SIZE and meta.buf_index <= BUF_SIZE){ apply(table_recode_r);
      } else { if(meta.buf_index == 1) { apply(table_recode_1);
      } else { if(meta.buf_index == 2) { apply(table_recode_2);
      } else { if(meta.buf_index == 3) { apply(table_recode_3);
      ...
      }}}}
    }
    else { if(packetType.label == TYPE_ACK and flowID.label == FLOW_ID and  genID.label == meta.gen_current) {

      apply(table_step_to_next_gen);
}}}}

control egress {
  // declared and left empty on purpose
}
```

The **table_load_1** (Listing 4.26 - line 10) matches on either a ACK or DATA packet, by validating if the *pkt.type*, *pkt.flowID* and *pkt.genID* were properly extracted, and verifying if *pkt.flowID* equals the constant FLOW_ID. In that case, it loads the *nc_enabled_flag* and *gen_current_flag*, which trigger the remaining execution of the pipeline.

Also, the logical if-else-if statements control the tables applied to DATA packets (Listing 4.26 - line 18) or to ACK packets (Listing 4.26 - line 34). As a result, the remaining tables require only a single default action without parameters.

Just like with the P4-XOR Switch (§ 4.3), we present the pipeline with two objectives. Firstly, to overview the existing tables and to have an idea of which of the algorithm's steps are within each table. Secondly, and most importantly, to contextualize the placement and sequence of tables mainly involved with the core functionality of the Buffering (§ 4.4.1), Coding (§ 4.4.2) and Forwarding (§ 4.4.3) Modules.

4.4.1 Buffering Module

Again, the required data structure comes directly from design (§ 3.3.3). First, we define the constants (Listing 4.27), which are then used for the declaration of metadata (Listing 4.28) and registers (Listing 4.29) alike. The number of declared payload register buffers is controlled by PAY_SIZE and the number of coefficient register buffers by GEN_SIZE.

Listing 4.27: Constants (P4-RLNC)

```
#define GF_BITS    256
#define GF_BYTES  8
#define GF_MOD    255

#define GEN_SIZE  3
#define PAY_SIZE  2
#define BUF_SIZE  10
```

Listing 4.28: Metadata (P4-RLNC)

```
header_type meta_t {
  fields {
    nc_enabled_flag : 1;

    buf_index : 32;
    buf_index_r : 32;

    gen_current : 32;
    gen_current_flag : 1;

    // ... omitted ...
    }
}
metadata meta_t meta;
```

Listing 4.29: Registers (P4-RLNC)

```
// Flags / Pointers
register nc_enabled        { width: 1;   instance_count: 1; }
register buf_index         { width: 32; instance_count: 1; }
register gen_current       { width: 32; instance_count: 1; }
register gen_current_flag { width: 1;   instance_count: 1; }

// Payload Buffers
register buf_p1 {
  width: GF_BYTES;
  instance_count: BUF_SIZE; }
register buf_p2 {
  width: GF_BYTES;
  instance_count: BUF_SIZE; }

// Coefficient Buffers
register buf_c1 {
  width: GF_BYTES;
  instance_count: BUF_SIZE; }
register buf_c2 {
  width: GF_BYTES;
  instance_count: BUF_SIZE; }
register buf_c3 {
  width: GF_BYTES;
  instance_count: BUF_SIZE; }
```

To store the contents of a DATA packet, two tables are used (Listing 4.30). If the buffers are not full, **table_write** is applied; else it's **table_overwrite**.

Listing 4.30: The Tables

```
table table_write {
  actions {
    action_write;
  }
}

table table_overwrite {
  actions {
    action_overwrite;
  }
}
```

Listing 4.31: The Compound Actions

```
action action_write () {
  register_read(meta.buf_index, buf_index, 0);
  register_write(buf_c1, meta.buf_index, coef[0].label);
  register_write(buf_c2, meta.buf_index, coef[1].label);
  register_write(buf_c3, meta.buf_index, coef[2].label);
  register_write(buf_p1, meta.buf_index, msg[0].content);
  register_write(buf_p2, meta.buf_index, msg[1].content);
}
action action_overwrite () {
  modify_field_rng_uniform(meta.buf_index_r, 0, meta.buf_index−1);
  // ... (the same as action_write(), therefore omitted) ...
}
```

Listing 4.32: The Table Population

```
table_set_default   table_write       action_write
table_set_default   table_overwrite   action_write
```

The respective actions (Listing 4.31) obtain the index of the column on which the *pkt.payload* (extracted as symbol's elements to the *msg* header stack, during parsing) and *pkt.coefs* are stored over the payload and coefficient buffers. There is only one difference. The **action_write** simply loads to metadata the *buf_index* of the next free column, persisted in a register. The **action_overwrite** additionally calculates and uses a random *buf_index_r* ranging from the first (given by 0) and the last columns (given by *buf_index* - 1).

As already stated, only DATA packets will reach this stage in the pipeline, as all the validations and verifications were previously made. Therefore, the two tables are populated exclusively with their respective action, as default (Listing 4.32).

4.4.2 Coding Module

The implementation of the Coding Module was not trivial and was one of the toughest challenges of this dissertation. Recalling from § 3.3.4, the coding model is a **Sparse RLNC with Incremental Density**. Therefore, we need a table for each density degree (Listing 4.34), which is populated with a single default action that starts the recoding process (Listing 4.36).

For each chosen packet (i.e., to be obtained from the buffers) for the recoding process, there must be PAY_SIZE metadata fields (Listing 4.33) for its symbol's elements and GEN_SIZE for the coefficients. As the maximum recoding density is also dictated by GEN_SIZE, we need metadata fields up to a GEN_SIZE number of packets.

In order to fully load a packet to metadata, there is a loading action per packet (Listing 4.35). Notice that these actions are not directly applied from a table match, but rather within the action configured for that exact same match itself (e.g., **action_recode_1**). Therefore, these loading actions do not take their parameters from the matches configured from a runtime CLI interface, implying that we are not constrained to the problem of not being able to pass metadata fields as parameters (as seen in § 4.3.1). As such, it's true that there could be a single compound action, taking the metadata fields identifiers as parameters. However, it would result in more complex code and a slight decrease in performance (we performed some tests to verify this).

Listing 4.33: Data Structures

```
// packet 1 payload          // packet 2 payload
p1_1 : GF_BYTES;             p2_1 : GF_BYTES;
p1_2 : GF_BYTES;             p2_2 : GF_BYTES;

// packet 1 coefficients     // packet 2 coefficients
c1_1 : GF_BYTES;             c2_1 : GF_BYTES;
c1_2 : GF_BYTES;             c2_2 : GF_BYTES;
c1_3 : GF_BYTES;             c2_3 : GF_BYTES;
```

Listing 4.35: Loading Packet to Metadata

```
action action_load_to_pkt_1 (idx) {
  register_read(meta.c1_1, buf_c1, idx);
  register_read(meta.c1_2, buf_c2, idx);
  register_read(meta.c1_3, buf_c3, idx);

  register_read(meta.p1_1, buf_p1, idx);
  register_read(meta.p1_2, buf_p2, idx);
}
```

Listing 4.34: The Tables

```
table table_recode_r  { actions { action_recode_r; } }
table table_recode_1 { actions { action_recode_1; } }
table table_recode_2 { actions { action_recode_2; } }
table table_recode_3 { actions { action_recode_3; } }
```

Listing 4.36: The Tables Population

```
table_set_default table_recode_r  action_recode_r
table_set_default table_recode_1 action_recode_1
table_set_default table_recode_2 action_recode_2
table_set_default table_recode_3 action_recode_3
```

To be more detailed, we review the recoding process, considering that GEN_SIZE equals 3 and PAY_SIZE equals 2. As an example, listing 4.37 depicts the recoding process for a density degree of 2. It starts by getting, and loading to metadata, a random coefficient vector (with length equal to the density), within $GF(2^8)$. Then, it also loads the necessary number of packets to metadata. Finally, it recodes the current packet, by updating its payload and coefficient vector.

The recoding process for a full density degree (Listing 4.38) is identical. The only difference is that we have to invoke the Fisher-Yates Shuffle Algorithm **(action_rng_init)** to randomly determine, without repetition, which packets are loaded to metadata.

Recoding is a complex and endeavor task. Not only because it requires $GF(2^8)$ arithmetic, but also due to the amount of symbol's elements and coefficients that each operation has as input, and the intermediate results that must be carried around. The same applies for the implementation of the Fisher-Yates Shuffle Algorithm. It is not trivial. Both tasks require a determined, and somewhat complicated, set of auxiliary structures and compound actions. Implementation details of each, are given below.

Listing 4.37: Recode Action: Density 2

```
action action_recode_2 () {

  // Get Random Coefficient Vector
  modify_field_rng_uniform(meta.rng_c1, 0,
                              GF_MOD);
  modify_field_rng_uniform(meta.rng_c2, 0,
                              GF_MOD);

  // Load Packets to Metadata
  action_load_to_pkt_1(0);
  action_load_to_pkt_2(1);

  // Update packet's PAYLOAD
  action_GF_mult_2(
    meta.rng_c1, meta.p1_1,
    meta.rng_c2, meta.p2_1
  );
  action_GF_add_1(
    meta.mult_result_1,
    meta.mult_result_2
  );
  modify_field(msg[0].content, meta.add_result_1);

  action_GF_mult_2(
    meta.rng_c1, meta.p1_2,
    meta.rng_c2, meta.p2_2
  );
  action_GF_add_1(
    meta.mult_result_1,
    meta.mult_result_2
  );
  modify_field(msg[1].content, meta.add_result_1);

  // Update packet's COEFFICIENTS
  action_GF_mult_2(
    meta.rng_c1, meta.c1_1,
    meta.rng_c2, meta.c2_1
  );
  action_GF_add_1(
    meta.mult_result_1,
    meta.mult_result_2
  );
  modify_field(coef[0].label, meta.add_result_1);

  action_GF_mult_2(
    meta.rng_c1, meta.c1_2,
    meta.rng_c2, meta.c2_2
  );
  action_GF_add_1(
    meta.mult_result_1,
    meta.mult_result_2
  );
  modify_field(coef[1].label, meta.add_result_1);

  action_GF_mult_2(
    meta.rng_c1, meta.c1_3,
    meta.rng_c2, meta.c2_3
  );
  action_GF_add_1(
    meta.mult_result_1,
    meta.mult_result_2
  );
  modify_field(coef[2].label, meta.add_result_1);
}
```

Listing 4.38: Recode Action: Full Density

```
action action_recode_r () {
  ... omitted ...
  // Load Packets to Metadata
  action_rng_init();
  action_load_to_pkt_1(meta.rng_result_1);
  action_load_to_pkt_2(meta.rng_result_2);
  action_load_to_pkt_3(meta.rng_result_3);
  ... omitted ...
}
```

Galois Field Arithmetic

Building up from the design phase (§ 3.3.4), we create two arrays of registers to persist the LOG *(GF256_log)* and ANTILOG *(GF256_invlog)* tables. As $GF(2^8)$ is considered, each cell needs 8 bits to represent the maximum value within the field (GF_BYTES). Recall from § 3.3.4 that the LOG table requires an entry of each field value (GF_BITS) and that the ANTILOG table is four times larger for optimization purposes. Notice how we explicitly set the array length (i.e., instance_count) for the ANTILOG table. This is because P4 does not allow arithmetic operations here (Listing 4.39).

These tables are pre-computed considering the primitive polynomial $2^8 + 2^4 + 2^3 + 2^2 + 1$ (i.e., 285) and are loaded to the respective arrays of registers, during the P4-RLNC switch startup, via the runtime CLI interface (Listing 4.40).

Listing 4.39: Data Structures

```
// The LOG table
register GF256_log {
  width: GF_BYTES;
  instance_count: GF_BITS; }

// The ANTILOG table
register GF256_invlog {
  width: GF_BYTES;
  instance_count: 1025; }  // GF_BITS*4 + 1

header_type meta_t {
  fields {
    // ... omitted ...

    // Addition: one result.
    add_result : GF_BYTES;

    // Multiplication: one result
    log1   : GF_BYTES;
    log2   : GF_BYTES;
    invlog  : GF_BYTES;

    // Addition: cumulative .
    add_result_1 : GF_BYTES;
    add_result_2 : GF_BYTES;

    // Multiplication: Pairs
    mult_result_1 : GF_BYTES;
    mult_result_2 : GF_BYTES;
    mult_result_3 : GF_BYTES;

    // ... omitted ...
  }
}
metadata meta_t meta;
```

Listing 4.40: LOG/ANTILOG Table Population

```
register_write  GF256_invlog  0  1
register_write  GF256_invlog  1  2
// ... omitted ...
register_write  GF256_log     0  512
register_write  GF256_log     1  255
// ... omitted ...
```

Listing 4.41: GF Addition: Cumulative Loop

```
action action_GF_add_1 (x1,x2) {
  action_GF_add(x1,x2);
  modify_field(meta.add_result_1,meta.add_result);
}
action action_GF_add_2 (x1,x2,x3) {
  action_GF_add_1 (x1,x2);
  action_GF_add     (meta.add_result_1,x3);
  modify_field(meta.add_result_2,meta.add_result);
}
```

Listing 4.42: GF Multiplication: Pairing Loop

```
action action_GF_mult_1 (x1,x2) {
  action_GF_mult(x1,x2);
  modify_field(meta.mult_result_1,meta.invlog);
}
action action_GF_mult_2 (x1,x2,x3,x4) {
  action_GF_mult_1 (x1,x2);
  action_GF_mult     (x3,x4);
  modify_field(meta.mult_result_2,meta.invlog);
}
action action_GF_mult_3 (x1,x2,x3,x4,x5,x6) {
  action_GF_mult_2 (x1,x2,x3,x4);
  action_GF_mult     (x5,x6);
  modify_field(meta.mult_result_3,meta.invlog);
}
```

Listing 4.43: GF Addition and GF Multiplication Arithmetic Operations

```
action action_GF_add (a,b) {
  bit_xor(meta.add_result, a, b);
}
action action_GF_mult (a,b) {
  register_read(meta.log1, GF256_log, a);
  register_read(meta.log2, GF256_log, b);
  register_read(meta.invlog, GF256_invlog, meta.log1 + meta.log2);
}
```

Bare in mind that GF operations are performed in pairs. Listing 4.43 shows how addition **(action_GF_add)** and multiplication **(action_GF_mult)** are implemented. Addition is the simple application of the built-in bit XOR primitive action over the two parameters. Multiplication uses the parameters to index and get two values from the LOG table, which are added and used as index to get the final result from the ANTILOG table. The results are kept in metadata for later reference (Listing 4.39).

Notice that recoding a single symbol's element of the current packet, requires several pairs of multiplications and then the cumulatively adding of all the multiplication results. Metadata fields to carry all these intermediary results are required (Listing 4.39).

However, modifying **action_GF_add** and **action_GF_mult** to take an additional parameter representing the target metadata field for each arithmetic operation result would not only be impractical, but also result in even more extensive and complex code.

Therefore, we came with a more compact and generic solution. Given the maximum recode density (consequence of GEN_SIZE), we define a cumulative addition cascade loop (Listing 4.41) and a pairing multiplication cascade loop (Listing 4.42).

For instance, recoding with a density degree of 3, would take three input packets. Therefore, for each symbol's element to be recoded, or coefficient to be updated, first we have 3 pairs of multiplications **(action_GF_mult_3)**. For the first symbol's element of the current packet: we multiply the first symbol's element of the first input packet by the first randomly chosen coefficient; the first element of the second packet by the second random coefficient; and so on. The methodology for updating the coefficients is the same. Then, the 3 pairs are cumulatively added **(action_GF_add_2)**.

Concluding, the depth of the loops is determined by the recoding density. To recode with density of 2, **action_GF_mult_2** and **action_GF_add_1** would be used instead, and so on and so forth.

Choosing Random Packets

Following from § 3.3.4, we implemented the Fisher-Yates Shuffle Algorithm. The data structure (Listing 4.44) consists on an array of registers *(rng_array)* that holds all the possible buffer indexes (i.e., from 0 to BUF_SIZE - 1) that might be chosen; metadata fields for the randomly chosen index *(rng_idx_rng)*, ranging from 0 to the last index *(rng_idx_max)* for the current iteration, which values are to be swapped *(rng_num_at_idx, rng_num_at_max)*; and finally metadata fields for the result of each iteration *(rng_result_1, rng_result_2, ...)*. Their bit-width equals the minimum required bits to represent BUF_SIZE.

To initialize the algorithm, the array is filled with the indexes to be randomly picked, and the last index for the first iteration is set (i.e., BUF_SIZE - 1).

Listing 4.44: Data Structures

```
register rng_array {
  width: 4;
  instance_count: BUF_SIZE;
}
header_type meta_t {
  fields {
    ...
    rng_result_1: 4;
    rng_result_2: 4;
    rng_result_3: 4;

    rng_idx_max: 4;
    rng_idx_rng: 4;

    rng_num_at_idx: 4;
    rng_num_at_max: 4;
    ...
  }
}
metadata meta_t meta;
```

Listing 4.45: Algorithm Initialization

```
action action_rng_init () {

 // Create array for indexes [ 0, BUF_SIZE [
 register_write(rng_array, 0 , 0);
 register_write(rng_array, 1 , 1);
 register_write(rng_array, 2 , 2);
 register_write(rng_array, 3 , 3);
 register_write(rng_array, 4 , 4);
 register_write(rng_array, 5 , 5);
 register_write(rng_array, 6 , 6);
 register_write(rng_array, 7 , 7);
 register_write(rng_array, 8 , 8);
 register_write(rng_array, 9 , 9);

 modify_field(meta.rng_idx_max,
              meta.buf_index - 1);

 action_rng_random_3();
}
```

Listing 4.46: Algorithm Execution Loop

```
action action_rng_random_1 () {
 action_rng_random_idx();
 action_rng_swap();
 register_read(meta.rng_result_1, rng_array,
              meta.rng_num_at_max);
 action_rng_update_max();
}
action action_rng_random_2 () {
 action_rng_random_idx();
 action_rng_swap();
 register_read(meta.rng_result_2, rng_array,
              meta.rng_num_at_max);
 action_rng_update_max();

 action_rng_random_1();
}
action action_rng_random_3 () {
 // ... (omitted) ...
 action_rng_random_2();
}
```

Listing 4.47: Fisher-Yates Shuffle Algorithm

```
action action_rng_random_idx () {
 modify_field_rng_uniform(meta.rng_idx_rng, 0,
                          meta.rng_idx_max);
}
action action_rng_swap () {
 register_read(meta.rng_num_at_idx, rng_array,
              meta.rng_idx_rng);
 register_read(meta.rng_num_at_max, rng_array,
              meta.rng_idx_max);

 register_write(rng_array, meta.rng_idx_rng,
               meta.rng_num_at_max);
 register_write(rng_array, meta.rng_idx_max,
               meta.rng_num_at_idx);
}
action action_rng_update_max () {
 modify_field(meta.rng_idx_max,
              meta.rng_idx_max - 1);
}
```

The core of the algorithm (Listing 4.47), is composed of four steps. First, a random index is picked within the range of the current iteration **(action_rng_random_idx)**. Then, the values of the array (i.e., the pretended indexes themselves) at the random index and last index are swapped **(action_rng_swap)**. The new value at the last index represents the final result for the current iteration, thus is saved to metadata. Finally, the last index is updated for the next iteration **(action_rng_update_max)**.

As for the implementation of Galois Field Arithmetic (§ 4.4.2), we create a loop for the algorithm, with the same reasons (Listing 4.46). As this algorithm is called when the payload and coefficient buffers are full, which results in full density recoding, then the loop depth is dictated by GEN_SIZE. For instance, if GEN_SIZE equals 3, the algorithm must be executed three times. Therefore, there are three actions that are called in cascade.

4.4.3 Forwarding Module

If by default, we intended to multicast all traffic on the remaining ports of the switch, we could implement something similar to the P4-XOR Switch (§ 4.3.3). However, we provide an example when a packet is just forwarded to a single port.

The table (Listing 4.48) matches on the *ingress_port* of the current packet. If there is a match, the forward action itself (Listing 4.49) sets the outgoing port by changing the *egress_spec* with the value received as parameter.

Listing 4.50 provides an example of populating a table to forward incoming packets from logical port 1 to logical port 2, and vice versa. Also, it is configured to drop incoming packets from other logical ports.

Listing 4.48: The Table

```
table table_forward {
  reads {
    standard_metadata.ingress_port : exact;
  }
  actions {
    _drop;
    action_forward;
  }
}
```

Listing 4.49: The Compound Action

```
action action_forward(port) {
  modify_field(standard_metadata.egress_spec, port);
}
```

Listing 4.50: The Table Population

```
table_set_default  table_forward  _drop
table_add          table_forward  action_forward  1 => 2
table_add          table_forward  action_forward  2 => 1
```

This is the behavior that we ended up establishing for the P4-RLNC switch, due to the topology that was used, during the Evaluation Phase of this dissertation, for the P4-RLNC Switch.

4.5 Summary

Throughout this chapter we detailed our implementation, alongside the faced challenges and restrictions, which led to certain implementation's decisions and choices. In particular, after detailing the headers (§ 4.1) and parsers (§ 4.2), we overviewed the buffering (§ 4.3.1, § 4.4.1), coding (§ 4.3.2, § 4.4.2) and forwarding (§ 4.3.3, § 4.4.3) modules, for both the P4-XOR Switch (§ 4.3) and the P4-RLNC Switch (§ 4.4).

Chapter 5 – Evaluation

Because the P4 language is fairly recent, the number of tools to evaluate P4 switches is very small, and those that exist are limited. As to the best of our knowledge, this is the first design and implementation of NC on P4-compliant devices, our main focus was on testing the correct functionality of our two solutions: P4-XOR Switch and P4-RLNC Switch.

In the case of the P4-XOR Switch, the evaluation was extended and performance was also analyzed. With that end in mind, we considered the butterfly network. This is an excellent testing environment due to the existent bottleneck, which showcases the performance improvement of an *interflow* solution.

Evaluating the performance of the P4-RLNC Switch is more difficult, mainly because the solution is *intraflow*, which requires a more complex testing environment. Nonetheless, being *intraflow* would not be the only reason to extend the evaluation beyond functionality tests. Remind that, RLNC grants great resilience against packet losses, node's failures and departures. However, due to the tight time schedule, the lack of P4-tools and system resources to setup and run the proper tests, the evaluation of the P4-RLNC Switch was not extended beyond functionality.

On the remaining of this chapter, we describe in more detail the experimental setup (§ 5.1), namely the network topologies, developed/used tools, and collected metrics. Following that, the setup and results of the functionality tests, for both the P4-XOR Switch (§ 5.2.1) and the P4-RLNC Switch (§ 5.2.2), are presented. Finally, we evaluate the performance of the P4-XOR Switch (§ 5.3).

5.1 Experimental Setup

We set up a VM running Ubuntu 16.10, with an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz and 4GB of RAM. Within this VM, we configured `Mininet` with the network topologies considered in the evaluation, to run the tests.

Mininet [60] is a network emulator that has been the main experimentation tool for SDN and P4-based work. The emulated network topologies consist of routers, switches, end-hosts and network links. The emulator runs on a single Linux Kernel. It uses lightweight virtualization (containers, each running in their own namespace) to make the system, on which it runs, look and behave similarly to a real network, running the same system, user code and kernel.

Instead of using Mininet's default switches, we run instances of the `simple_switch`, which is the reference P4-compliant target. To be able to run these switches, the P4 Consortium provides a python extension for Mininet, which enables the emulator to accept the .json file that results from the P4 program's compilation.

The network topologies we consider are depicted in figure 5.1.

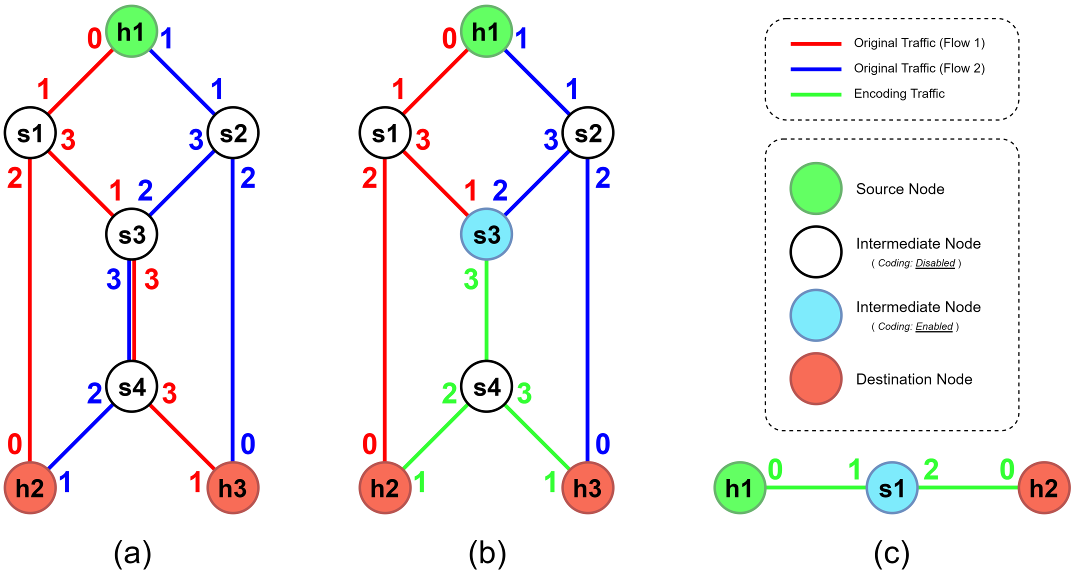


Figure 5.1: Testing Environment's Network Topologies **(a)** P4-XOR Switch: Performance **(b)** P4-XOR Switch: Functionality and Performance **(c)** P4-RLNC Switch: Functionality

To test the P4-XOR Switch we use the butterfly network topology. It consists of a source node (*h1*) with two logical ports. From each port, it sends traffic of a different flow. Destination nodes (*h2, h3*) will receive original traffic, from a single flow on their logical port 0, and encoding traffic (Fig. 5.1 (b)) or original traffic (Fig. 5.1 (a)) on logical port 1. Intermediate nodes (*s1, s2, s3, s4*) are loaded with the P4-XOR Switch program. Nodes *s1*, *s2* and *s3* have the Buffering and Coding modules disabled. Nodes *s1* and *s2* simply multicast all incoming traffic from logical port 1 on the remaining ports (logical ports 1 and 2).

The behavior of nodes *s3* and *s4* is not fixed. It is slightly changed depending on the test. For some performance tests (§ 5.3), the Buffering and Coding modules of *s3* are disabled. In that case, *s4* will receive original traffic on its logical port 1. Traffic from one flow (represented in red) is forwarded to logical port 3, and traffic from the other flow (represented in blue) to logical port 2 (see Fig. 5.1 (a)).

For the functionality tests (§ 5.2.1), the Buffering and Coding modules of *s3* are enabled, thus *s4* receives encoding traffic. Therefore, node *s4* multicasts all traffic to the remaining ports (see Fig. 5.1 (b)).

For the P4-RLNC Switch, we use a topology with a single source node (*h1*), intermediate node (*s1*) and destination node (*h2*). For each generation, *h1* sets the original symbols and then, sends DATA packets containing an encoded symbol alongside its global encoding vector. The node *s1* is loaded with the P4-RLNC Switch program, and performs Sparse RLNC with Incremental Density, over the incoming packets from logical port 0. Node *h2* receives and keeps the contents of DATA packets until it is able to decode the generation, which triggers sending an ACK packet back to *h1* in order to step to the next generation (see Fig. 5.1 (c)).

We mentioned at the beginning of this chapter the lack of resources to be a constraint to extending the P4-RLNC Switch evaluation beyond functionality. Just have an idea, if we were to consider a GEN_SIZE of 100 and a BUF_SIZE of 101, the P4 program would have more than 1 million lines of code. We even tried to load a P4 program with these parameters, on a topology with a single intermediate node. However, Mininet consumed all the resources of the VM, making it crash. This was one of the reasons why we restricted ourselves to functionality tests on a simple topology. As future work we plan to investigate ways to make the solution scale better.

The behavior of both source and destination nodes was implemented with python scripts. For packet generation we used the `Scapy` package.

Depending on the test, the network and encoding parameters need to be changed. Consequently, the P4 programs had to be adjusted accordingly. For convenience, and to ease this task, we created two java programs to generate the P4 programs themselves. For the P4-XOR Switch, we refer to it as `XORgenerator`, and for the P4-RLNC Switch, as `RLNCgenerator`.

Mininet provides a terminal prompt, where commands to view and/or change the network's state can be inserted. Xterm terminals may also be opened at any node, serving either as an host or a switch. This feature provides the means to run these python scripts and other network monitoring tools.

5.2 Functionality Tests

This section presents the functionality tests conducted of both the P4-XOR Switch and P4-RLNC Switch.

5.2.1 P4-XOR Switch

To test the functionality of the P4-XOR Switch, we considered two distinct flows. For each flow, we send a single packet with 8 bits of payload.

By using the XORgenerator, we generate the P4 program accordingly. The constants FLOW_ID_1 and FLOW_ID_2, are set to 500 and 501, respectively.

The source node *h1* starts by sending a packet with the ascii character *'a'* on logical port 0 (Fig. 5.2), and a packet with the ascii character *'b'* on logical port 1 (Fig. 5.3). We will refer to these as packet A and packet B.

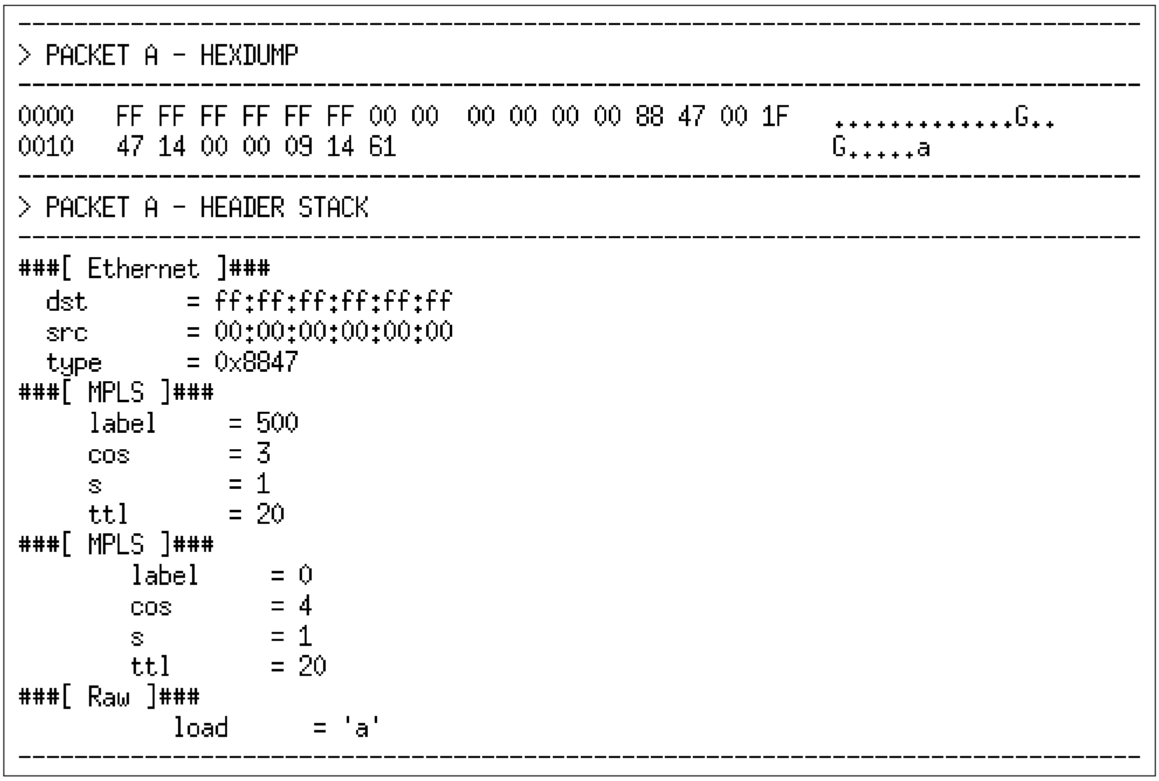
Intermediate node *s1* receives packet A and multicasts it on the remaining ports. As expected, destination node *h2* receives packet A on logical port 0 (Fig. 5.4).

Intermediate node *s2* receives packet B and multicasts it on the remaining ports. As expected, destination node *h3* receives packet B on logical port 0 (Fig. 5.6).

Intermediate node *s3* receives, stores and performs XOR over both packet A and packet B. Notice that the ascii character *'a'* corresponds to *'\x61'* in hexadecimal and the ascii character *'b'* to *'\x62'*. Therefore the XOR operation results in *'\x03'*. Let's call it packet AB. Finally, *s3* forwards packet AB to *s4*.

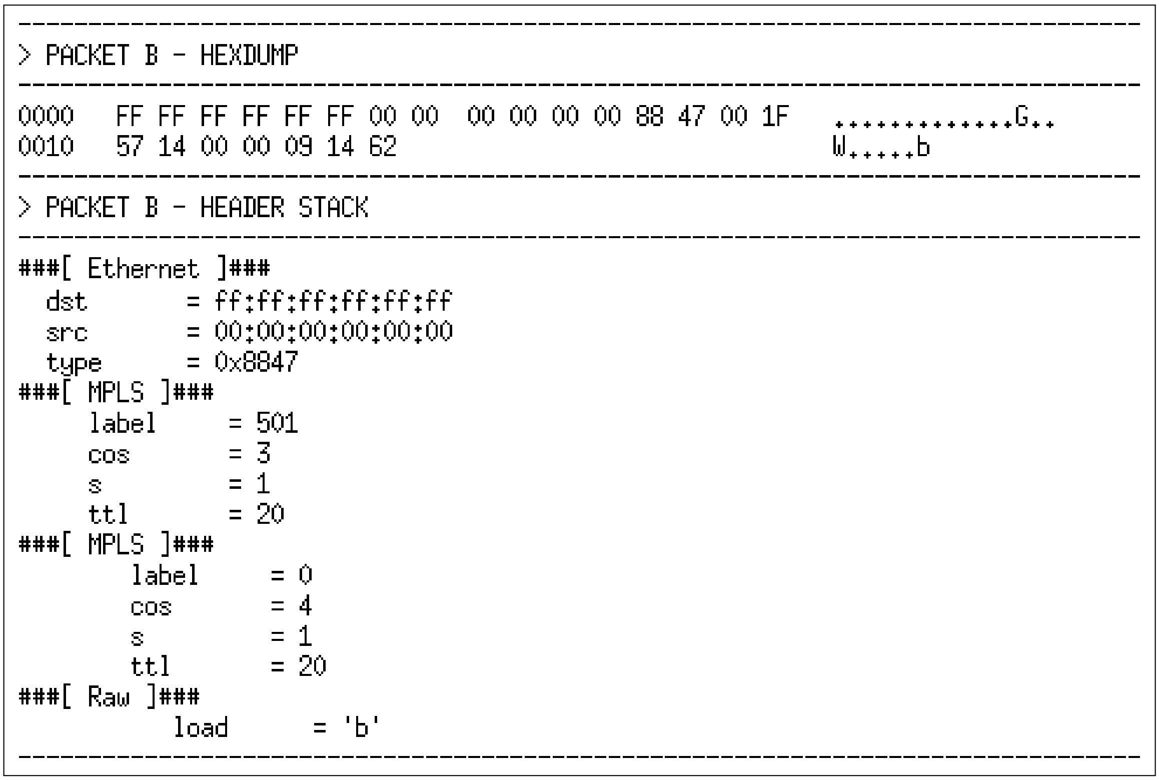
Intermediate node *s4* receives packet AB and multicasts it on the remaining ports. As expected, destination node *h2* receives the packet AB on logical port 1 (Fig. 5.5). The same applies for destination node *h3* (Fig. 5.7).

Concluding, the functionality of the P4-XOR Switch is demonstrated to be correct. First of all, notice from all the prints that the protocol header stack is correct. As the result was as expected, parsing was surely correct. Secondly, without a correct Forwarding Module, maybe the packet A and packet B would not reach destination nodes *h2* and *h3*, respectively. Also, maybe both would not reach intermediate node *s3*. This would result in packet AB not being created. Packet AB would also not exist without a correct Buffering and Coding Modules.



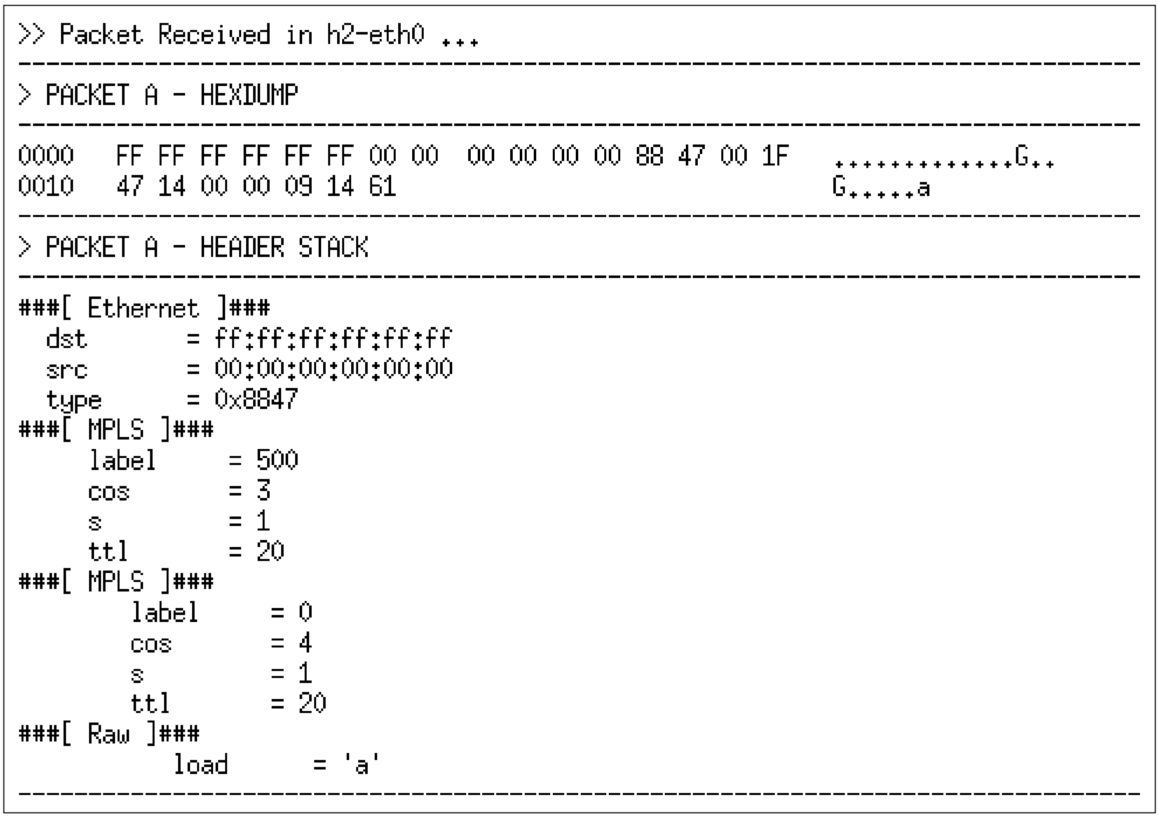
```
--------------------------------------------------------------------------------
> PACKET A - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   .............G..
0010   47 14 00 00 09 14 61                               G.....a
--------------------------------------------------------------------------------
> PACKET A - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 500
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = 'a'
--------------------------------------------------------------------------------
```

Figure 5.2: Sending report of packet A - Source node h1, logical port 0.



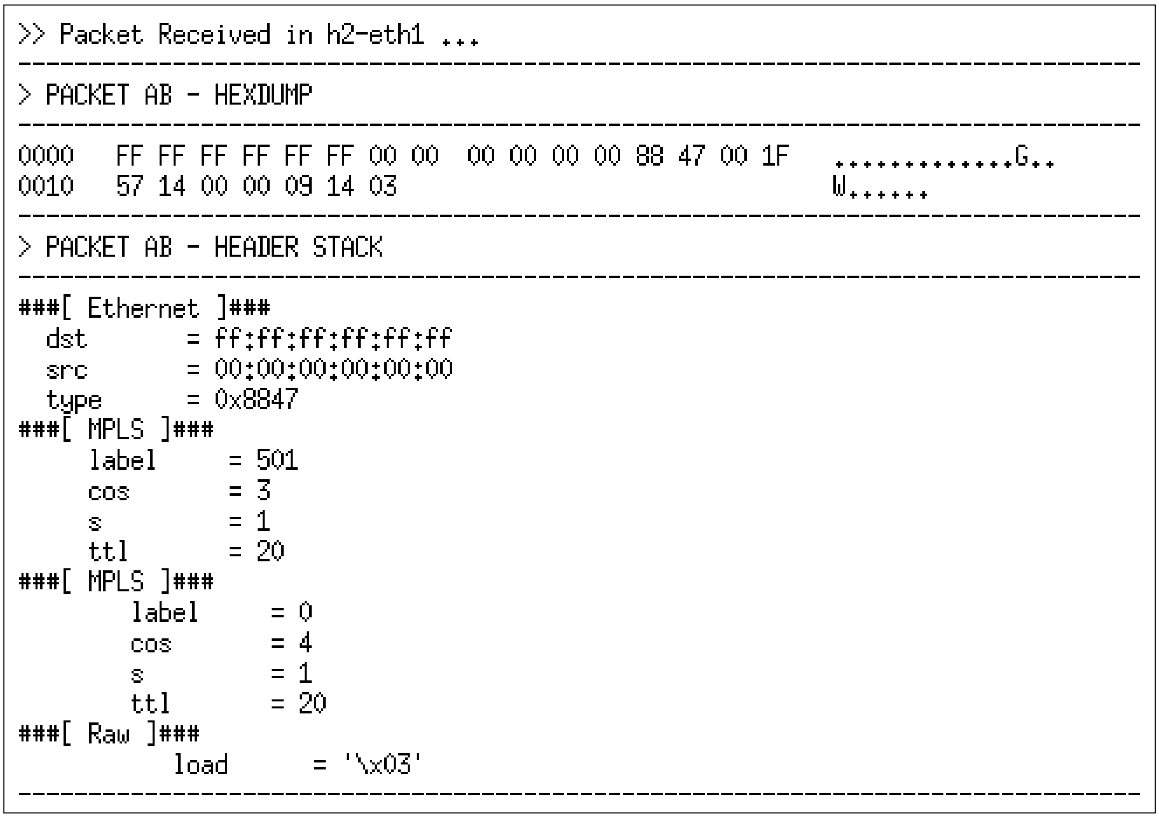
```
--------------------------------------------------------------------------------
> PACKET B - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   .............G..
0010   57 14 00 00 09 14 62                               W.....b
--------------------------------------------------------------------------------
> PACKET B - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 501
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = 'b'
--------------------------------------------------------------------------------
```

Figure 5.3: Sending report of packet B - Source node h1, logical port 1.



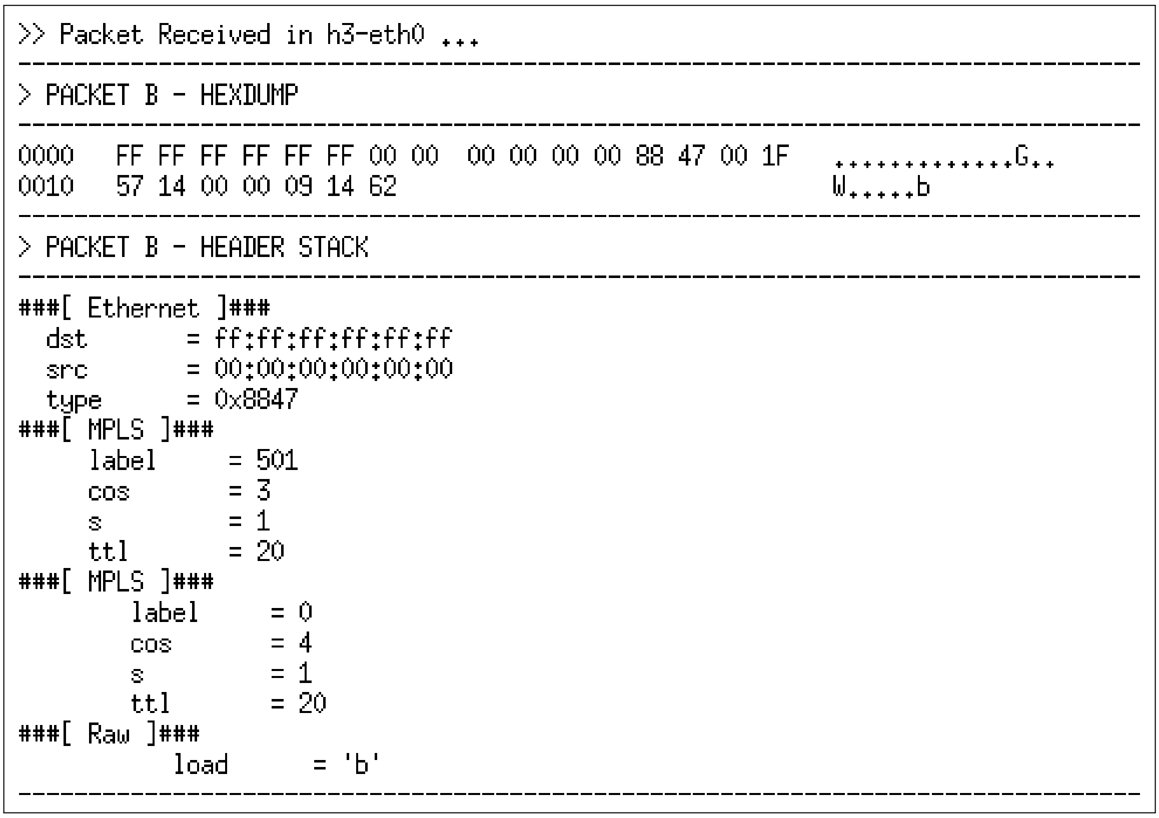
```
>> Packet Received in h2-eth0 ...
--------------------------------------------------------------------------------
> PACKET A - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   .............G..
0010   47 14 00 00 09 14 61                               G.....a
--------------------------------------------------------------------------------
> PACKET A - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 500
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = 'a'
--------------------------------------------------------------------------------
```

Figure 5.4: Reception report of packet A - Destination node h2, logical port 0.



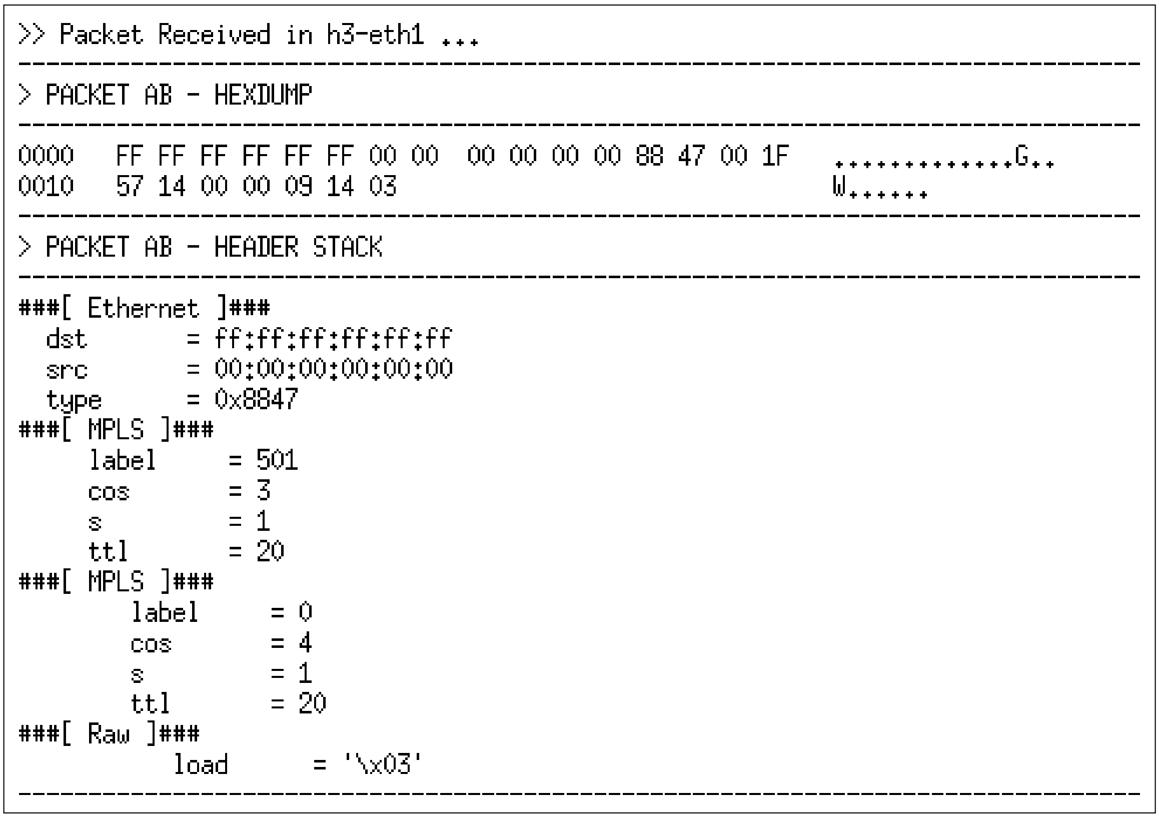
```
>> Packet Received in h2-eth1 ...
--------------------------------------------------------------------------------
> PACKET AB - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   .............G..
0010   57 14 00 00 09 14 03                               W......
--------------------------------------------------------------------------------
> PACKET AB - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 501
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = '\x03'
--------------------------------------------------------------------------------
```

Figure 5.5: Reception report of packet AB - Destination node h2, logical port 1.



```
>> Packet Received in h3-eth0 ...
--------------------------------------------------------------------------------
> PACKET B - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   ..............G..
0010   57 14 00 00 09 14 62                               W.....b
--------------------------------------------------------------------------------
> PACKET B - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 501
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = 'b'
--------------------------------------------------------------------------------
```

Figure 5.6: Reception report of packet B - Destination node h3, logical port 0.



```
>> Packet Received in h3-eth1 ...
--------------------------------------------------------------------------------
> PACKET AB - HEXDUMP
--------------------------------------------------------------------------------
0000   FF FF FF FF FF FF 00 00  00 00 00 00 88 47 00 1F   ..............G..
0010   57 14 00 00 09 14 03                               W......
--------------------------------------------------------------------------------
> PACKET AB - HEADER STACK
--------------------------------------------------------------------------------
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 00:00:00:00:00:00
  type      = 0x8847
###[ MPLS ]###
     label     = 501
     cos       = 3
     s         = 1
     ttl       = 20
###[ MPLS ]###
        label     = 0
        cos       = 4
        s         = 1
        ttl       = 20
###[ Raw ]###
           load      = '\x03'
--------------------------------------------------------------------------------
```

Figure 5.7: Reception report of packet AB - Destination node h3, logical port 1.

5.2.2 P4-RLNC Switch

To test the functionality of the P4-RLNC Switch, we follow the process of creating and encoding a whole generation at a source node (*h1*); recode it at an intermediate node (*s1*); and finally decode it at a destination node (*s2*).

In order to provide an example that can be fully visualized, we consider a generation size of 3 and a symbol size of 4. In other words, each generation is composed of 3 original symbols, where each symbol is divided in 4 elements. As always, we consider $GF(2^8)$.

The source and destination python scripts are adjusted accordingly. The P4 program as well, by using RLNCgenerator.

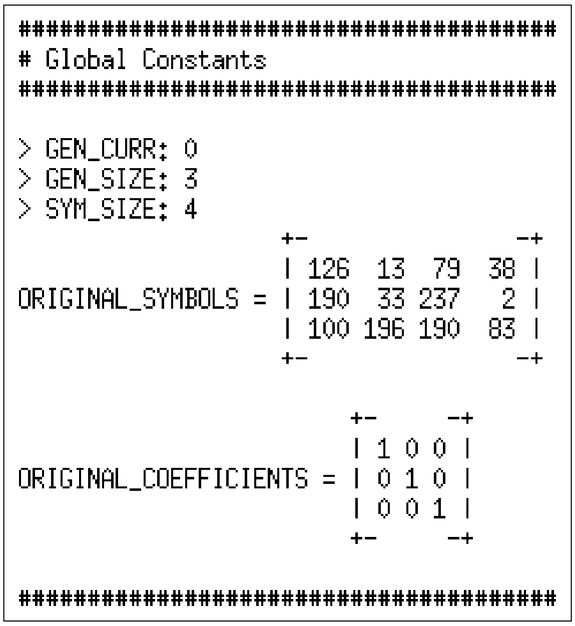
The source node *h1* starts by randomly creating the original symbols and respective original coefficient vectors (Fig. 5.8). Three DATA packets are sufficient to decode the generation. Therefore, *h1* encodes and sends three DATA packets (Figs. 5.9, 5.10, 5.11).

Notice that the procedure is the same for all the DATA packets. First a local encoding vector (LEV), within $GF(2^8)$ is obtained. Using LEV and the original symbols' elements as input to the encoding algorithm, the new symbol's elements are calculated. Also, using LEV and the original encoding vectors, the DATA packet's global encoding vector (GEV) is updated. Because the original encoding vectors are unitary, the resulting GEV equals LEV.

For each DATA packet received (Figs. 5.12, 5.13, 5.14), the destination node *h2* extracts, and organizes in matrices, the symbol's elements (ENCODED_SYMBOLS) and the respective global encoding vector (GLOBAL_ENCODING). Notice how the received DATA packets do not correspond to what was sent. This is because each DATA packet was recoded at intermediate node *s1*.

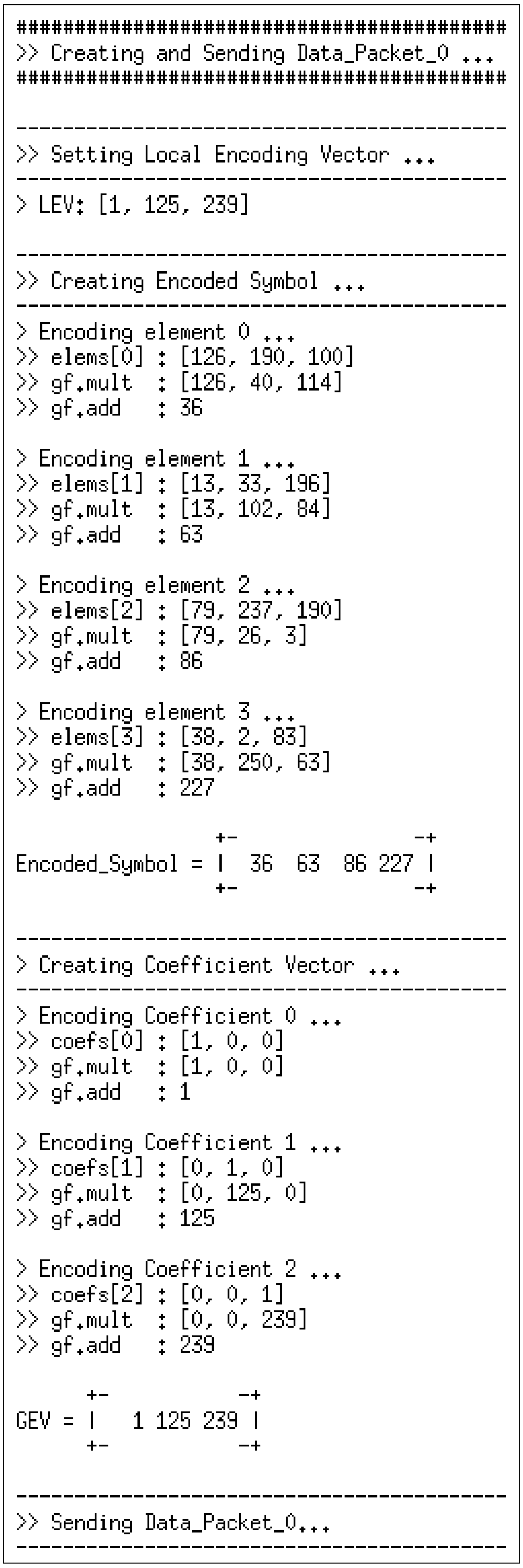
After receiving the three DATA packets, the rank of the ENCODED_SYMBOLS and GLOBAL_ENCODING matrices equals the generation size. This implies the generation can be decoded. From figure 5.14, we verify that the generation was decoded with success, as the ORIGINAL_SYMBOLS matrix equals the one from figure 5.8.

Concluding, as the DATA packets were recoded at the intermediate node *s1* with our P4 program, and the generation was successfully decoded, we demonstrate the correct functionality of the P4-RLNC Switch program.



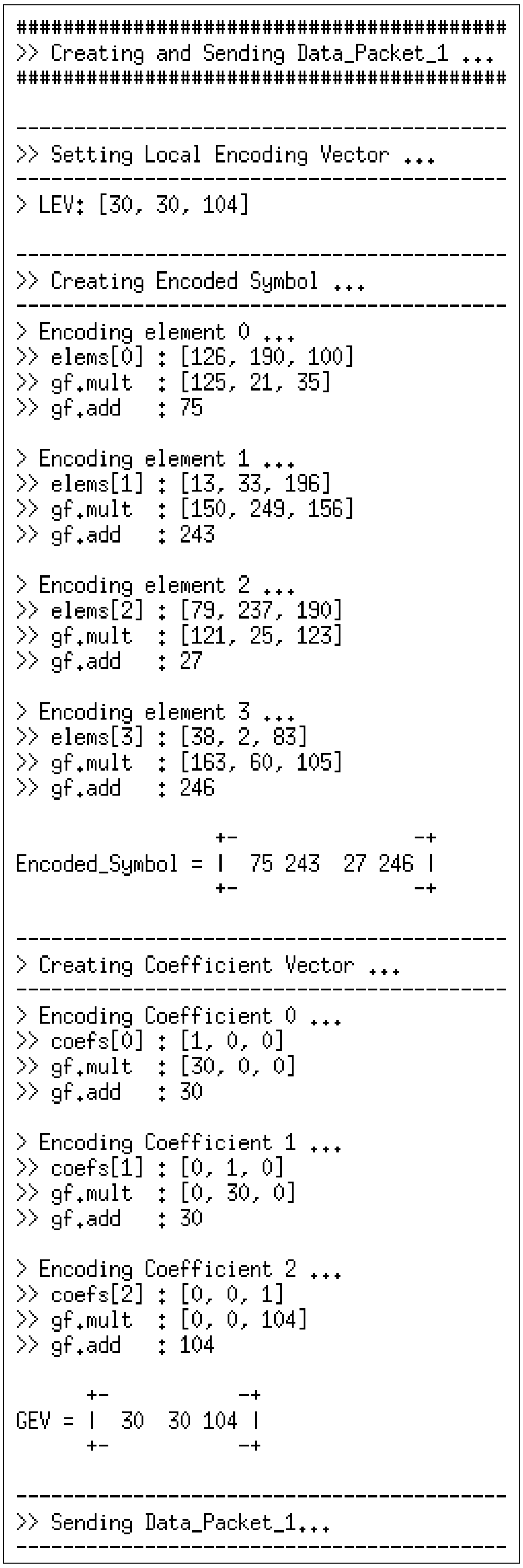
```
########################################
# Global Constants
########################################

> GEN_CURR: 0
> GEN_SIZE: 3
> SYM_SIZE: 4
                    +-              -+
                    | 126  13  79  38 |
ORIGINAL_SYMBOLS =  | 190  33 237   2 |
                    | 100 196 190  83 |
                    +-              -+

                         +-     -+
                         | 1 0 0 |
ORIGINAL_COEFFICIENTS =  | 0 1 0 |
                         | 0 0 1 |
                         +-     -+

########################################
```

Figure 5.8: Creation of a generation.



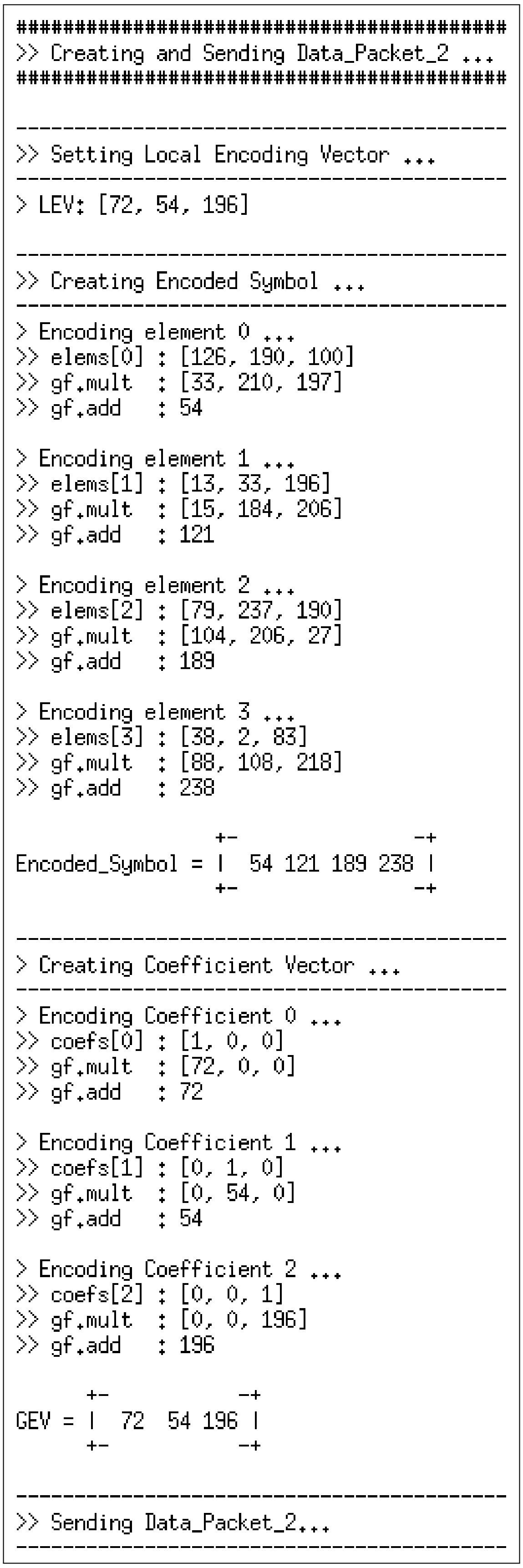
```
########################################
>> Creating and Sending Data_Packet_0 ...
########################################

----------------------------------------
>> Setting Local Encoding Vector ...
----------------------------------------
> LEV: [1, 125, 239]

----------------------------------------
>> Creating Encoded Symbol ...
----------------------------------------
> Encoding element 0 ...
>> elems[0] : [126, 190, 100]
>> gf.mult  : [126, 40, 114]
>> gf.add   : 36

> Encoding element 1 ...
>> elems[1] : [13, 33, 196]
>> gf.mult  : [13, 102, 84]
>> gf.add   : 63

> Encoding element 2 ...
>> elems[2] : [79, 237, 190]
>> gf.mult  : [79, 26, 3]
>> gf.add   : 86

> Encoding element 3 ...
>> elems[3] : [38, 2, 83]
>> gf.mult  : [38, 250, 63]
>> gf.add   : 227

                 +-               -+
Encoded_Symbol = |   36  63  86 227 |
                 +-               -+

----------------------------------------
> Creating Coefficient Vector ...
----------------------------------------
> Encoding Coefficient 0 ...
>> coefs[0] : [1, 0, 0]
>> gf.mult  : [1, 0, 0]
>> gf.add   : 1

> Encoding Coefficient 1 ...
>> coefs[1] : [0, 1, 0]
>> gf.mult  : [0, 125, 0]
>> gf.add   : 125

> Encoding Coefficient 2 ...
>> coefs[2] : [0, 0, 1]
>> gf.mult  : [0, 0, 239]
>> gf.add   : 239

      +-           -+
GEV = |    1 125 239 |
      +-           -+

----------------------------------------
>> Sending Data_Packet_0...
----------------------------------------
```

Figure 5.9: Creation of the first DATA packet.



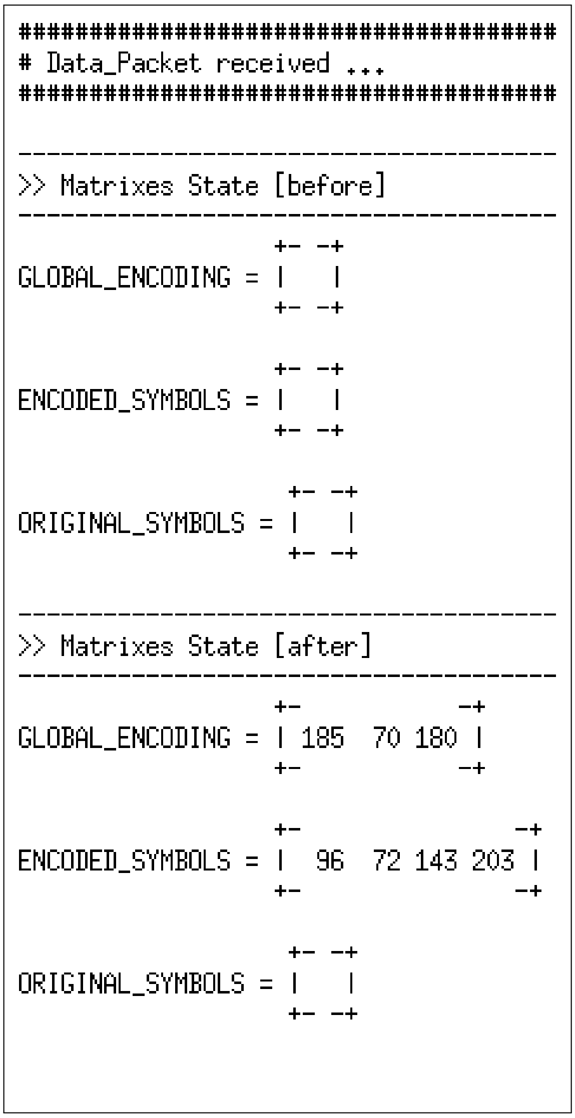
```
##########################################
>> Creating and Sending Data_Packet_1 ...
##########################################

------------------------------------------
>> Setting Local Encoding Vector ...
------------------------------------------
> LEV: [30, 30, 104]

------------------------------------------
>> Creating Encoded Symbol ...
------------------------------------------
> Encoding element 0 ...
>> elems[0] : [126, 190, 100]
>> gf.mult  : [125, 21, 35]
>> gf.add   : 75

> Encoding element 1 ...
>> elems[1] : [13, 33, 196]
>> gf.mult  : [150, 249, 156]
>> gf.add   : 243

> Encoding element 2 ...
>> elems[2] : [79, 237, 190]
>> gf.mult  : [121, 25, 123]
>> gf.add   : 27

> Encoding element 3 ...
>> elems[3] : [38, 2, 83]
>> gf.mult  : [163, 60, 105]
>> gf.add   : 246

                 +-                -+
Encoded_Symbol = |  75 243  27 246 |
                 +-                -+

------------------------------------------
> Creating Coefficient Vector ...
------------------------------------------
> Encoding Coefficient 0 ...
>> coefs[0] : [1, 0, 0]
>> gf.mult  : [30, 0, 0]
>> gf.add   : 30

> Encoding Coefficient 1 ...
>> coefs[1] : [0, 1, 0]
>> gf.mult  : [0, 30, 0]
>> gf.add   : 30

> Encoding Coefficient 2 ...
>> coefs[2] : [0, 0, 1]
>> gf.mult  : [0, 0, 104]
>> gf.add   : 104

      +-           -+
GEV = |  30  30 104 |
      +-           -+

------------------------------------------
>> Sending Data_Packet_1...
------------------------------------------
```

Figure 5.10: Creation of the second DATA packet.



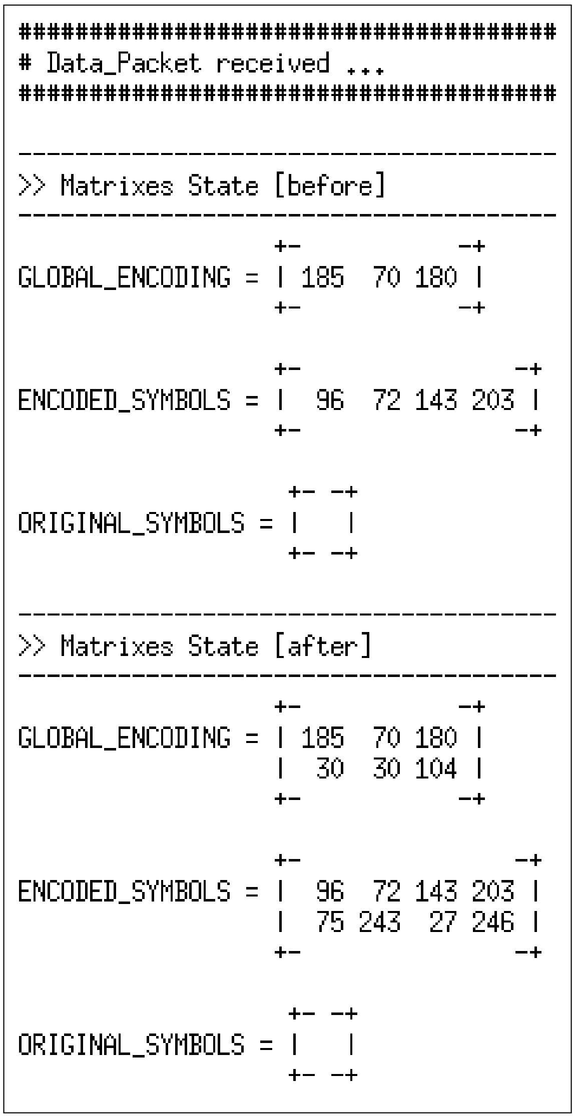
```
##########################################
>> Creating and Sending Data_Packet_2 ...
##########################################

------------------------------------------
>> Setting Local Encoding Vector ...
------------------------------------------
> LEV: [72, 54, 196]

------------------------------------------
>> Creating Encoded Symbol ...
------------------------------------------
> Encoding element 0 ...
>> elems[0] : [126, 190, 100]
>> gf.mult  : [33, 210, 197]
>> gf.add   : 54

> Encoding element 1 ...
>> elems[1] : [13, 33, 196]
>> gf.mult  : [15, 184, 206]
>> gf.add   : 121

> Encoding element 2 ...
>> elems[2] : [79, 237, 190]
>> gf.mult  : [104, 206, 27]
>> gf.add   : 189

> Encoding element 3 ...
>> elems[3] : [38, 2, 83]
>> gf.mult  : [88, 108, 218]
>> gf.add   : 238

                 +-                -+
Encoded_Symbol = |  54 121 189 238 |
                 +-                -+

------------------------------------------
> Creating Coefficient Vector ...
------------------------------------------
> Encoding Coefficient 0 ...
>> coefs[0] : [1, 0, 0]
>> gf.mult  : [72, 0, 0]
>> gf.add   : 72

> Encoding Coefficient 1 ...
>> coefs[1] : [0, 1, 0]
>> gf.mult  : [0, 54, 0]
>> gf.add   : 54

> Encoding Coefficient 2 ...
>> coefs[2] : [0, 0, 1]
>> gf.mult  : [0, 0, 196]
>> gf.add   : 196

      +-           -+
GEV = |  72  54 196 |
      +-           -+

------------------------------------------
>> Sending Data_Packet_2...
------------------------------------------
```

Figure 5.11: Creation of the third DATA packet.



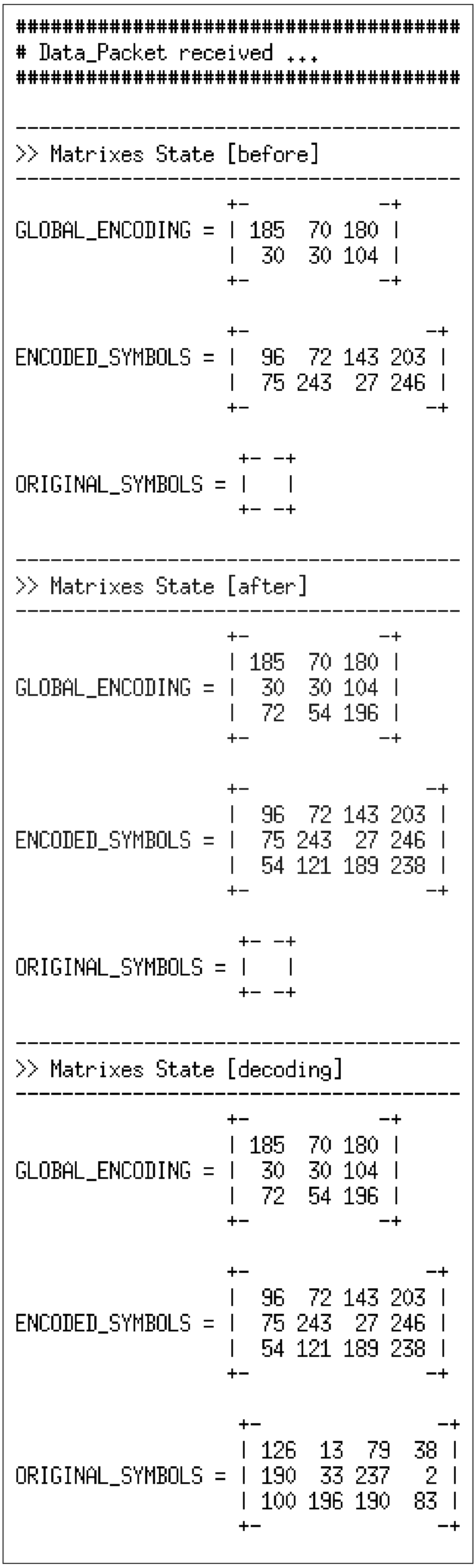
```
######################################
# Data_Packet received ...
######################################

--------------------------------------
>> Matrixes State [before]
--------------------------------------
                  +- -+
GLOBAL_ENCODING = |   |
                  +- -+

                  +- -+
ENCODED_SYMBOLS = |   |
                  +- -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+

--------------------------------------
>> Matrixes State [after]
--------------------------------------
                  +-           -+
GLOBAL_ENCODING = | 185  70 180 |
                  +-           -+

                  +-               -+
ENCODED_SYMBOLS = |  96  72 143 203 |
                  +-               -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+
```

Figure 5.12: Reception of the first recoded DATA packet.



```
######################################
# Data_Packet received ...
######################################

--------------------------------------
>> Matrixes State [before]
--------------------------------------
                  +-           -+
GLOBAL_ENCODING = | 185  70 180 |
                  +-           -+

                  +-               -+
ENCODED_SYMBOLS = |  96  72 143 203 |
                  +-               -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+

--------------------------------------
>> Matrixes State [after]
--------------------------------------
                  +-           -+
GLOBAL_ENCODING = | 185  70 180 |
                  |  30  30 104 |
                  +-           -+

                  +-               -+
ENCODED_SYMBOLS = |  96  72 143 203 |
                  |  75 243  27 246 |
                  +-               -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+
```

Figure 5.13: Reception of the second recoded DATA packet.



```
######################################
# Data_Packet received ...
######################################

--------------------------------------
>> Matrixes State [before]
--------------------------------------
                  +-           -+
GLOBAL_ENCODING = | 185  70 180 |
                  |  30  30 104 |
                  +-           -+

                  +-               -+
ENCODED_SYMBOLS = |  96  72 143 203 |
                  |  75 243  27 246 |
                  +-               -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+

--------------------------------------
>> Matrixes State [after]
--------------------------------------
                  +-           -+
                  | 185  70 180 |
GLOBAL_ENCODING = |  30  30 104 |
                  |  72  54 196 |
                  +-           -+

                  +-               -+
                  |  96  72 143 203 |
ENCODED_SYMBOLS = |  75 243  27 246 |
                  |  54 121 189 238 |
                  +-               -+

                   +- -+
ORIGINAL_SYMBOLS = |   |
                   +- -+

--------------------------------------
>> Matrixes State [decoding]
--------------------------------------
                  +-           -+
                  | 185  70 180 |
GLOBAL_ENCODING = |  30  30 104 |
                  |  72  54 196 |
                  +-           -+

                  +-               -+
                  |  96  72 143 203 |
ENCODED_SYMBOLS = |  75 243  27 246 |
                  |  54 121 189 238 |
                  +-               -+

                   +-               -+
                   | 126  13  79  38 |
ORIGINAL_SYMBOLS = | 190  33 237   2 |
                   | 100 196 190  83 |
                   +-               -+
```

Figure 5.14: Reception of the third recoded DATA packet.

5.3 Performance of the P4-XOR Switch

To evaluate the performance of the P4-XOR Switch, we ran a set of tests on the butterfly network, where two distinct flows were considered. Each test had a duration of ten minutes and is briefly described as follows. On each of its logical ports, the source node *h1* injects traffic of a certain flow, at a specific rate. The rate is the same for both flows. At the same time, we collect metrics (detailed below) and compute their average and standard deviation.

To be more detailed, the set of tests consists of injecting traffic at several different rates, in two distinct scenarios.

The first scenario is depicted in Figure 5.1 (a). The Buffering and Coding Modules of all intermediate nodes (*s1, s2, s3, s4*) were disabled. Therefore, packets are exclusively forwarded such that both flows can reach the destination nodes *h2* and *h3* **(Scenario 1)**.

The second scenario is depicted in Figure 5.1 (b). The Buffering and Coding Modules of the intermediate node *s3* were enabled. Therefore, *s3* will transfer a single encoded flow, instead of two original flows **(Scenario 2)**.

The metrics collected were: **Throughput** and **CPU Utilization**. To measure the throughput, we count the number of received packets, per second, on all interfaces of both destination nodes *h2* and *h3*. We also measure the CPU utilization to verify if (or, at which point), the throughput values are bound by the CPU. From the start, we expected that the performance would be CPU bounded eventually, as we are running the tests on a VM, within a modest laptop. To collect the metrics, we used the `NMON Visualizer`.

Table 5.1 summarizes the rates of injected traffic that we worked with, and the expected theoretical throughput for all the interfaces of the destination nodes *h2* and *h3*, in both scenarios.

Injected Traffic (pkt/s) Node Interface	1	10	25	50	100	150	200	250	300	400	500	750	1000
h2-eth0 (Scenario 1)	1	10	25	50	100	150	200	250	300	400	500	750	1000
h2-eth0 (Scenario 2)	1	10	25	50	100	150	200	250	300	400	500	750	1000
h2-eth1 (Scenario 1)	0.5	5	12.5	25	50	75	100	125	150	200	250	375	500
h2-eth1 (Scenario 2)	1	10	25	50	100	150	200	250	300	400	500	750	1000
h3-eth0 (Scenario 1)	1	10	25	50	100	150	200	250	300	400	500	750	1000
h3-eth0 (Scenario 2)	1	10	25	50	100	150	200	250	300	400	500	750	1000
h3-eth1 (Scenario 1)	0.5	5	12.5	25	50	75	100	125	150	200	250	375	500
h3-eth1 (Scenario 2)	1	10	25	50	100	150	200	250	300	400	500	750	1000

Table 5.1: Expected Throughput on Destination Node's Interfaces, depending on the Injected Traffic and Scenario.

As we can see, this testing environment showcases the advantage of XOR in the presence of bottlenecks in a network. Namely, in scenario 2 the average throughput is expected to be equal to the rate of the injected traffic, at all interfaces. This is because, due to XOR, there are no bottlenecks. Each link exclusively carries either a single original flow or a single encoded flow.

For scenario 1 the same does not apply. Intermediate node *s3* receives two original flows, and as there is no XOR, it must forward both, on a single interface. Therefore, there is a bottleneck on the link between intermediate nodes *s3* and *s4*. As a consequence the average throughput at the logical ports 0 (i.e. h2-eth0 and h3-eth0) is expected to be half of the injected traffic's rate.

On an important note, in order to create the bottleneck between *s3* and *s4*, the bandwidth of all the links has to be properly configured. In our case, it needs to be equal to the injected traffic's rate. If not, and for instance, the injected traffic's rate was 50 pkt/s, *s3* would forward 100 pkt/s. On the same manner, *s4* would also receive and forward 100 pkt/s. Consequently, destination nodes would receive on their logical port 1, 50 pkt/s in scenario 1 and 100 pkt/s in scenario 2, which is the double of the expected values.

A PAYLOAD_SIZE of 8912 and BUFFER_SIZE of 1000 were used for all tests. These values were chosen to load the network conveniently. We had them fixed, so that the results would not be influenced by chance, if these parameters were to be changed among tests.

Summarizing, for each test we setup the butterfly topology within Mininet, adjust the bandwidth of the links to the injected traffic's rate of the traffic generator, and use the XORgenerator, to create the P4 program accordingly. Finally, we run the traffic generator for 10 minutes, and collect the metrics during that period.

5.3.1 CPU Utilization

The measurements of the CPU utilization are depicted in figure 5.15. We review this first, in order to have a better understanding, later on, of the throughput measurements.

As we can see, our system can handle, approximately, a rate of injected traffic ranging between 100 and 150 pkt/s, given the butterfly topology and two distinct flows. In other words, for rates up to 150 pkt/s the throughput measurements will correspond to the expected theoretical ones. For higher rates, the performance is CPU bound and thus, the throughput measurements will not have the required fidelity [60].

However, we have to consider that the standard deviation, ranging between the rates 25 pkt/s and 200 pkt/s is significant. This implies that for a considerable period of the duration of the tests, the percentages were higher or lower than the average. This is relevant,

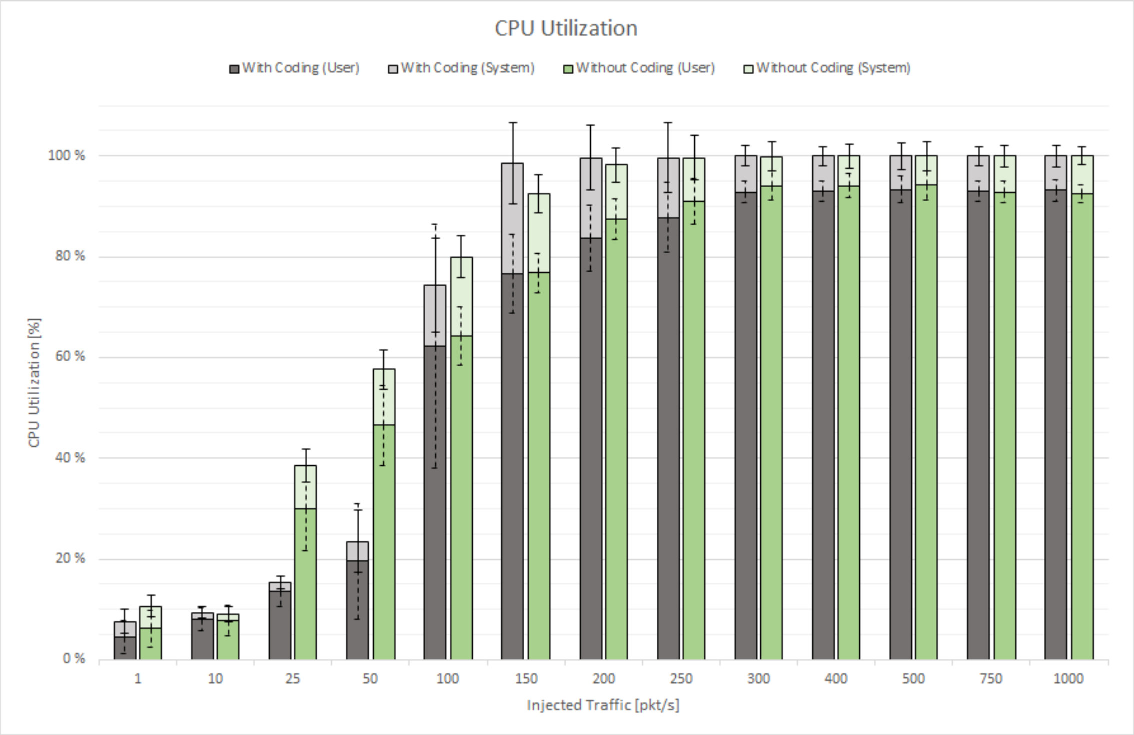


Figure 5.15: Measured CPU utilization, depending on the Injected Traffic and Scenario.

specially for the rates from 150 pkt/s to 200 pkt/s, where the average gets close to 100%.

It is interesting to note that a scenario without coding tends to use, and requires more CPU. There might be two reasons for this. The first reason, is due to the Buffering and Coding modules being disabled at intermediate node *s3*. Therefore, its intern queues/buffers between the ingress and egress pipelines fill faster, using more resources. The second reason, is related to the fact that intermediate node *s3*, has to forward two original flows, instead of a single encoded flow. As a consequence, more packets are discarded and the network is more congested. We conjecture these to be the main reasons for more resources being used.

Concluding, XOR not only improves the overall performance of a network with bottlenecks, but also requires less resources.

5.3.2 Throughput

We organized the throughput measurements in four bar graphs, one for each interface (or, logical port). Each graph represents the results, for each injected traffic's rate, in both scenarios.

The results for logical ports 0, are depicted in Figures 5.16 and 5.18. As expected, up to a injected traffic's rate of 150 pkt/s, the average throughput corresponds to the theoretical values. For the scenario with XOR enabled, the results are slightly lower for rates of 100 pkt/s and 150 pkt/s, but by taking a closer look at the standard deviation, we can see that the throughput reaches 100 pkt/s and 150 pkt/s respectively, at certain times. We anticipate to be due to the lack of fidelity from these values upwards. The Mininet paper [60] explicitly mentions 50% of CPU load as a threshold to guarantee emulation fidelity.

Remind that, for rates higher than 150 pkt/s, the CPU utilization reaches 100%, thus the results are not completely trustworthy. By interpreting them nonetheless, this implies that the system can't handle all the traffic, which results on an average throughput lower than expected. However, from the standard deviation for rates 250 pkt/s and 300 pkt/s, we can observe that the expected theoretical throughput can be achieved. From higher rates, we can conclude that the throughput can reach approximately a maximum of 260 pkt/s, on average. The standard deviation tells us that the absolute maximum throughput is in the order of 315 pkt/s.

The results for logical ports 1, are depicted in Figures 5.17 and 5.19. It is clearly evident that XOR doubles the throughput. Again, and as expected, up to an injected traffic's rate of 150 pkt/s, the average throughput corresponds to the theoretical values. For higher rates, when the results are bounded by the CPU, the standard deviation shows that the theoretical throughput values might still be reached for rates up to 250 pkt/s, at certain time periods. From rates beyond 250 pkt/s, the average and maximum throughputs show evidence to be approximately the same ones as for the logical ports 0.

Concluding, given our system and network parameters, our solution can correctly handle throughputs ranging between 150 pkt/s and 200 pkt/s on average, without provoking congestion and drops of packets. When the network starts to drop packets and is congested, the maximum achievable throughput is in the order of 250 pkt/s, on average.

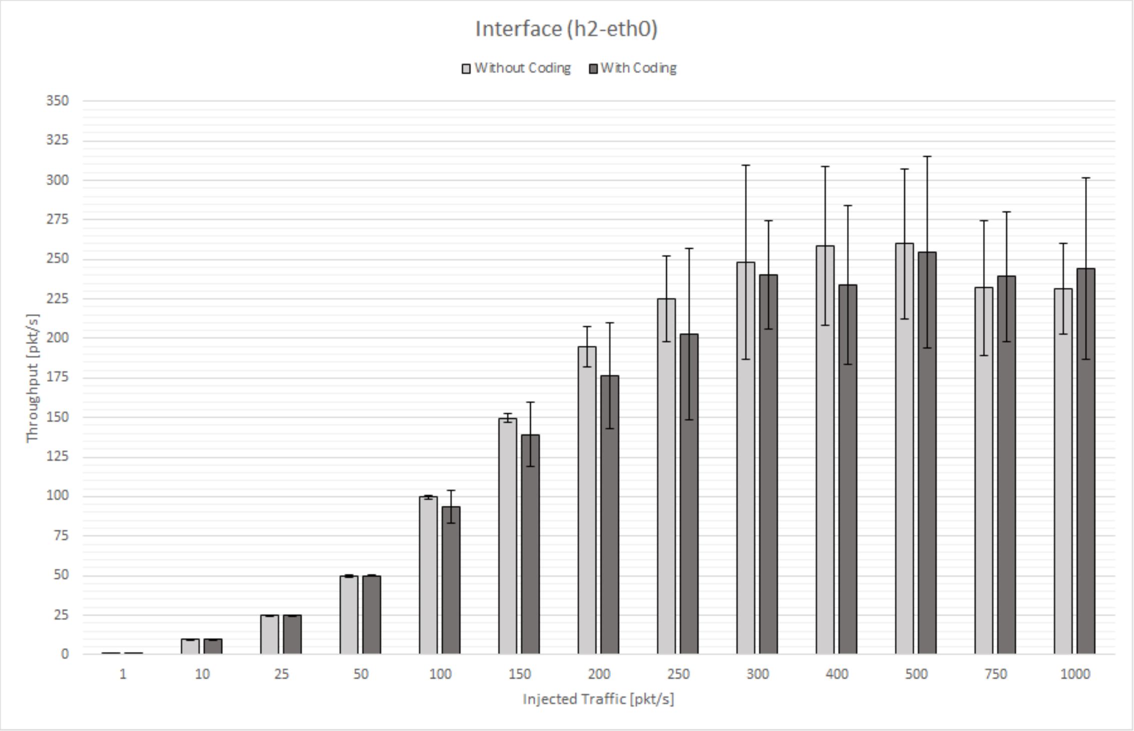


Figure 5.16: Measured Throughput on logical port 0 of destination node h2, depending on the Injected Traffic and Scenario.

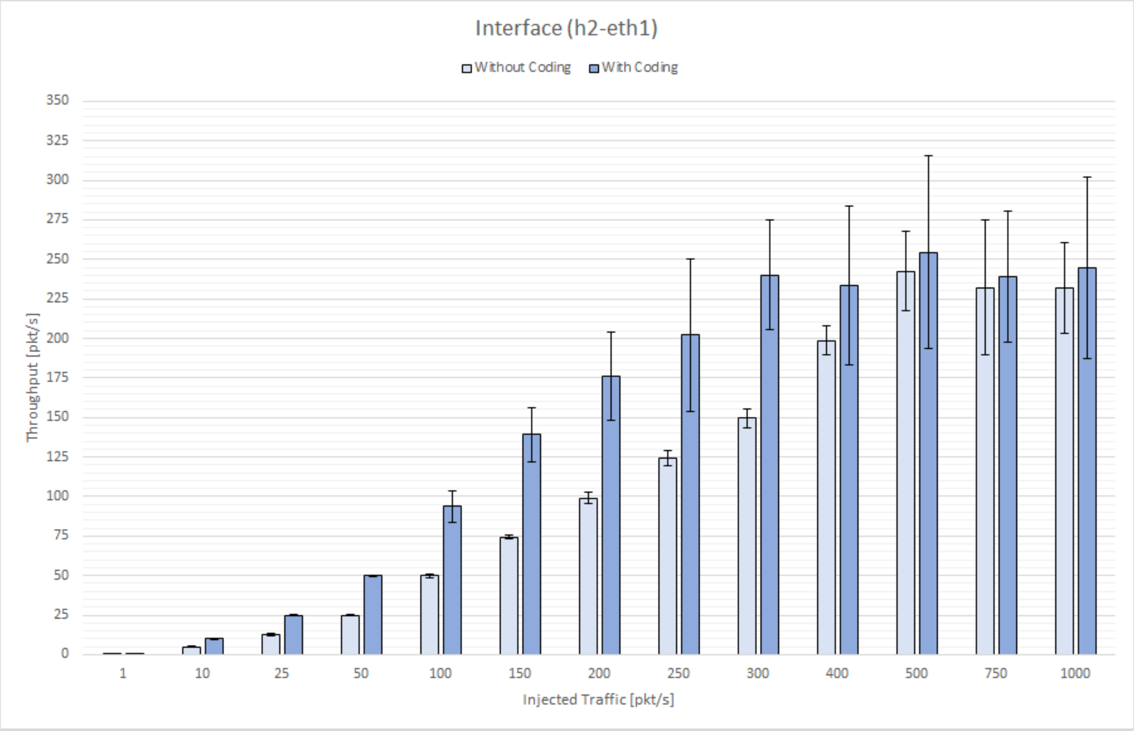


Figure 5.17: Measured Throughput on logical port 1 of destination node h2, depending on the Injected Traffic and Scenario.

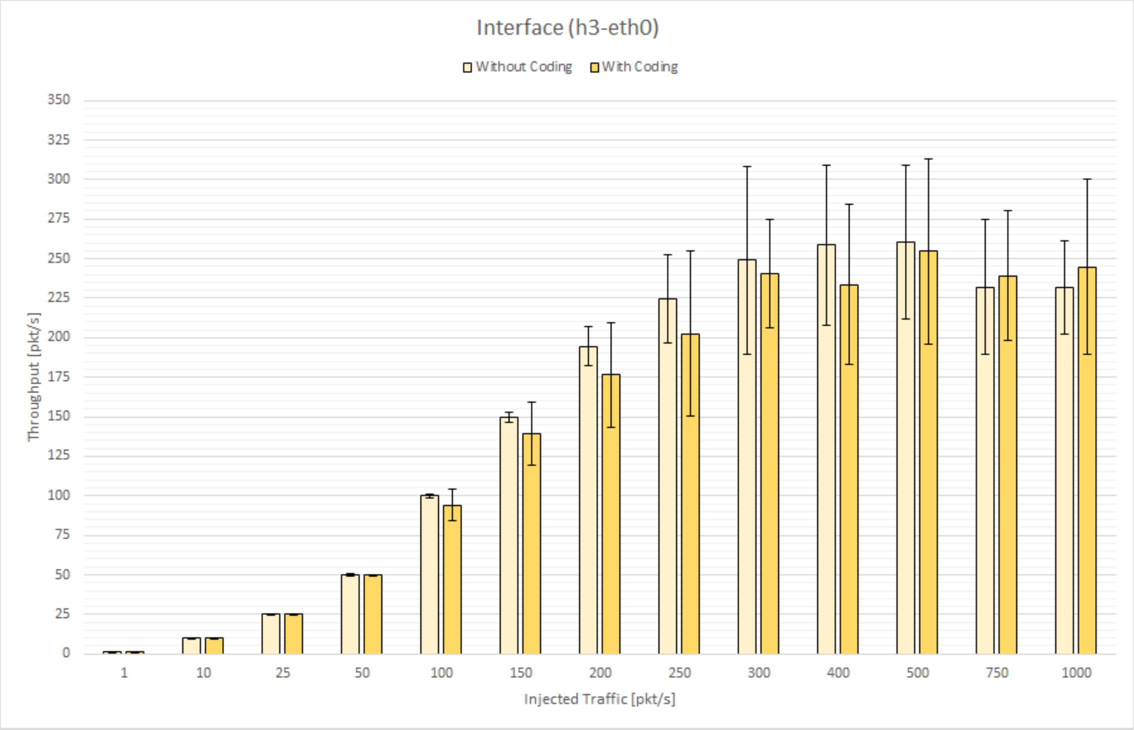


Figure 5.18: Measured Throughput on logical port 0 of destination node h3, depending on the Injected Traffic and Scenario.

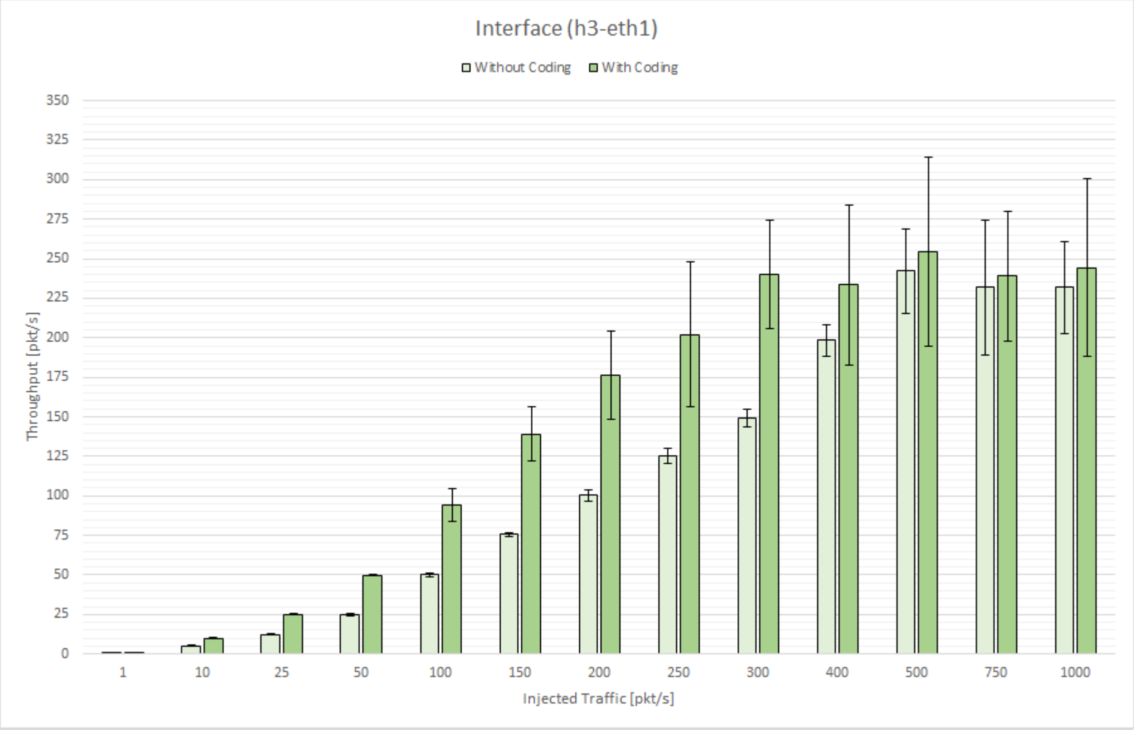


Figure 5.19: Measured Throughput on logical port 1 of destination node h3, depending on the Injected Traffic and Scenario.

5.4 Summary

In this chapter, we began by describing the experimental setup and overviewing the developed/used tools (§ 5.1). Then, we evaluated the functionality (§ 5.2) of both the P4-XOR Switch (§ 5.2.1) and the P4-RLNC Switch (§ 5.2.2). Additionally, we also evaluated the performance (§ 5.3) of the P4-XOR Switch, by collecting CPU (§ 5.3.1), and throughput (§ 5.3.2) metrics.

Chapter 6 – Conclusion & Future Work

In this dissertation, our goal was to design and implement a switch, capable of performing NC within the data plane, for the first time. For that purpose, we used a new promising high-level language for switch programming, known as P4.

Prior to the design and implementation of our solution, the fundamental concepts behind both NC and P4 were explored. We concluded that linear codes (LNC) were the best way to take advantage of how packets transverse a network, in order to improve the throughput and robustness. Yet, LNC considers a fixed set of local encoding vectors across all intermediate nodes. In order to find and provide a coding solution, a specific algorithm has to be used before starting the network. This not only binds the solution to a specific network topology, but also would render the design and implementation of our solution unnecessarily more complex. Thankfully, with random linear codes (RLNC), we learned that by choosing the local encoding vectors randomly over a sufficiently large galois field, each time a new packet is to be encoded, a coding solution is guaranteed with almost 100% probability.

However, after understanding the inner details of the P4 language, we verified that programming a NC switch would be difficult and challenging. The main reason is that the P4 language was not designed with focus on complex payload processing, which is intrinsic to NC. Moreover, buffers to store packets for future encoding opportunities would be necessary. We came to the realization that we had to implement our own buffers.

Thankfully, with an unusual, non-trivial and unorthodox use of the simplistic data structures and limited set of primitive actions of P4, we were able to achieve our goal with success. The hard work and time invested resulted in two solutions: P4-XOR Switch and P4-RLNC Switch.

The P4-XOR Switch was the starting point, and consisted of a simple XOR linear code. The implementation was relatively straightforward as P4 offers a primitive action to perform XOR offer two different sets of bits.

The P4-RLNC Switch consisted of a custom RLNC variant, of our own design. When a recoding opportunity arises, all packets are used (incremental density), until there are enough packets for full recoding (in this case, a fixed number of packets is selected without repetition). This time around, the implementation was more challenging. Mainly, because the P4 language does not support cycles. As such, in order to deal with the substantial number of operations required to perform linear combinations, we created a series of generic actions that call each other in cascade.

Finally, the evaluation we conducted validated the functionality of both switches. Furthermore, we showcased how the P4-XOR Switch improves over traditional routing, in a butterfly topology.

In conclusion, not only did we successfully implemented two NC data planes, but proved that with a bit of imagination, and work around, the P4 language has the potential to define non trivial kinds of packet processing.

As future work, the current solution could be migrated to P4-16, which is the most recent version and expected to be the future reference. Note that moving from P4-14 to P4-16 is not direct nor trivial as the abstract forwarding model that was used in P4-14 has experienced a radical change in P4-16, alongside the language itself, whose design has suffered a significant overhaul. As we understood across this dissertation, the P4-14 version has the limitation of not directly supporting payload processing. In P4-16, extern objects could help easing this task as they give freedom to the programmer to build external modules to be used by P4. It would also be interesting to envision and implement new encoding schemes.

To finish, the evaluation was performed in software, recurring to a network emulator - Mininet. In order to further understand the potential gains and limitations of our solutions, the next step would be to extend the evaluation to hardware switches, such as Barefoot Tofino.

Glossary

API Application Programming Interface.

CPU Central Processing Unit.

GF Galois Field.

LNC Linear Network Coding.

NC Network Coding.

PP Primitive Polynomial.

RLNC Random Linear Network Coding.

SDN Software Defined Network.

Bibliography

[1] Barefoot Tofino. https://barefootnetworks.com/technology/. Accessed 10-February-2017.

[2] Brocade SDN Controller. http://www.brocade.com/en/possibilities/technology/sdn-and-opendaylight.html. Accessed 10-January-2017.

[3] Dell OpenFlow enabled switches. http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Force10_S4810_Spec_sheet.pdf. Accessed 10-January-2017.

[4] Hewlett-Packard OpenFlow enabled switches. https://www.hpe.com/us/en/networking/sdn.html#infrastructure. Accessed 10-January-2017.

[5] Huawei SDN Enabled Routers. http://pr.huawei.com/en/news/hw-193480-sdn.htm#.WKEdfTuLTIU. Accessed 10-January-2017.

[6] Intel Ethernet Switch Silicon FM6000 Series. http://www.intel.com/content/www/us/en/ethernet-products/switch-silicon/ethernet-switch-fm5000-fm6000-series.html. Accessed 20-December-2016.

[7] Mellanox SwitchX SX1016. http://www.mellanox.com/related-docs/prod_eth_switches/PB_SX1016.pdf. Accessed 10-January-2017.

[8] NEC Genesis Hosting Solutions: Case Studies. http://www.nec.com/en/case/genesis/. Accessed 18-January-2017.

[9] NEC Internal Data Center: Case Studies. http://www.nec.com/en/case/idc/. Accessed 18-January-2017.

[10] NEC Kanazawa University Hospital: Case Studies. http://www.nec.com/en/case/kuh/. Accessed 19-January-2017.

[11] NEC ProgrammableFlow Networking. https://www.necam.com/sdn/. Accessed 10-January-2017.

[12] ONF. Open Networking Foundation. https://www.opennetworking.org. Accessed 20-December-2016.

[13] OpenFlow @ Google. In Open Networking Summit. http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf. Accessed 18-January-2017.

[14] OpenFlow Switch Specification, Version 1.1.0. http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf. Accessed 20-September-2016.

[15] P4-14 Language Specification, Version 1.0.4. https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed 28-May-2017.

[16] P4-16 Language Specification, Version 1.0.3. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-p4-lang-def. Accessed 28-May-2017.

[17] P4 adoption continues to grow rapidly. http://p4.org/technical-steering-committee/p4-adoption-continues-to-grow-rapidly/. Accessed 23-February-2017.

[18] P4 Behavioral Model 2. https://github.com/p4lang/behavioral-model. Accessed 15-October-2016.

[19] P4 Compiler. https://github.com/p4lang/p4c. Accessed 15-October-2016.

[20] P4 Consortium. http://p4.org. Accessed 20-September-2016.

[21] P4 Consortium Blog. http://p4.org/blog/. Accessed 20-September-2016.

[22] Pica8 Switches. http://www.pica8.com/open-switching/1-gbe-10gbe-open-switches.php. Accessed 10-January-2017.

[23] Software-Defined Networking: Next-Gen Enterprise Networks. In Open Networking Summit. http://www.opennetsummit.org/archives/apr12/davy-wed-enterprise.pdf. Accessed 19-January-2017.

[24] Rudolf Ahlswede, Ning Cai, S-YR Li, and Raymond W Yeung. Network information flow. *IEEE Transactions on information theory*, 46(4):1204–1216, 2000.

[25] Riccardo Bassoli, Hugo Marques, Jonathan Rodriguez, Kenneth W Shum, and Rahim Tafazolli. Network coding theory: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1950–1978, 2013.

[26] 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.

[27] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 99–110. ACM, 2013.

[28] Szymon Chachulski, Michael Jennings, Sachin Katti, and Dina Katabi. More: A network coding approach to opportunistic routing. 2006.

[29] Szymon Chachulski, Michael Jennings, Sachin Katti, and Dina Katabi. Trading structure for randomness in wireless opportunistic routing. 37(4), 2007.

[30] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[31] Cheng-Chih Chao, Ching-Chun Chou, and Hung-Yu Wei. Pseudo random network coding design for ieee 802.16 m enhanced multicast and broadcast service. In *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*, pages 1–5. IEEE, 2010.

[32] Philip A Chou and Yunnan Wu. Network coding for the internet and wireless networks. *IEEE Signal Processing Magazine*, 24(5):77–85, 2007.

[33] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(1):18–24, 2016.

[34] Douglas SJ De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. *Wireless networks*, 11(4):419–434, 2005.

[35] Jonathan Detchart, Emmanuel Lochin, Jérôme Lacan, and Vincent Roca. Tetrys, an on-the-fly network coding protocol. 2015.

[36] Alexandros G Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[37] Muhammad Zubair Farooqi, Salma Malik Tabassum, Mubashir Husain Rehmani, and Yasir Saleem. A survey on network coding: From traditional wireless networks to emerging cognitive radio networks. *Journal of Network and Computer Applications*, 46:166–181, 2014.

[38] Soheil Feizi, Daniel E Lucani, Chres W Sørensen, Ali Makhdoumi, and Muriel Médard. Tunable sparse network coding for multicast networks. In *Network Coding (NetCod), 2014 International Symposium on*, pages 1–6. IEEE, 2014.

[39] Victor Firoiu, Greg Lauer, Brian DeCleene, and Soumendra Nanda. Experiences with network coding within manet field experiments. In *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*, pages 1363–1368. IEEE, 2010.

[40] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[41] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[42] Christos Gkantsidis and Pablo Rodriguez Rodriguez. Network coding for large scale content distribution. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, pages 2235–2245. IEEE, 2005.

[43] Nicholas JA Harvey, David R Karger, and Kazuo Murota. Deterministic network coding by matrix completion. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 489–498. Society for Industrial and Applied Mathematics, 2005.

[44] Janus Heide, Morten V Pedersen, Frank HP Fitzek, and Muriel Médard. A perpetual code for network coding. In *Vehicular Technology Conference (VTC Spring), 2014 IEEE 79th*, pages 1–6. IEEE, 2014.

[45] M Néstor J Hernández, Morten V Pedersen, Péter Vingelmann, Janus Heide, Daniel E Lucani, and Frank HP Fitzek. Getting kodo: Network coding for the ns-3 simulator. In *Proceedings of the Workshop on ns-3*, pages 101–107. ACM, 2016.

[46] Tracey Ho, Ralf Koetter, Muriel Medard, David R Karger, and Michelle Effros. The benefits of coding over routing in a randomized setting. 2003.

[47] Tracey Ho, Muriel Médard, Ralf Koetter, David R Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[48] Cheng Huang and Lihao Xu. Fast software implementation of finite field operations. *Washington University in St. Louis, Tech. Rep*, 2003.

[49] Sidharth Jaggi, Peter Sanders, Philip A Chou, Michelle Effros, Sebastian Egner, Kamal Jain, and Ludo MGM Tolhuizen. Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982, 2005.

[50] CHI Kaikai, Xiaohong Jiang, and Susumu Horiguchi. A more efficient cope architecture for network coding in multihop wireless networks. *IEICE transactions on communications*, 92(3):766–775, 2009.

[51] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: practical wireless network coding. In *ACM SIGCOMM computer communication review*, volume 36, pages 243–254. ACM, 2006.

[52] D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. 1969.

[53] Ralf Koetter and Muriel Médard. Beyond routing: An algebraic approach to network coding. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 122–130. IEEE, 2002.

[54] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Transactions on Networking (TON)*, 11(5):782–795, 2003.

[55] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[56] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[57] Jeppe Krigslund, Jonas Hansen, Daniel E Lucani, Frank HP Fitzek, and Muriel Médard. Network coded software defined networking: Design and implementation. In *European Wireless 2015; 21th European Wireless Conference; Proceedings of*, pages 1–6. VDE, 2015.

[58] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[59] Michael Langberg, Alexander Sprintson, and Jehoshua Bruck. Network coding: A computational perspective. *IEEE Transactions on Information Theory*, 55(1):147–157, 2009.

[60] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[61] Baochun Li and Di Niu. Random network coding in peer-to-peer networks: from theory to practice. *Proceedings of the IEEE*, 99(3):513–523, 2011.

[62] S-YR Li, Raymond W Yeung, and Ning Cai. Linear network coding. *IEEE transactions on information theory*, 49(2):371–381, 2003.

[63] Yunfeng Lin, Baochun Li, and Ben Liang. Efficient network coded data transmissions in disruption tolerant networks. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1508–1516. IEEE, 2008.

[64] Sicheng Liu and Bei Hua. Ncos: A framework for realizing network coding over software-defined network. In *Local Computer Networks (LCN), 2014 IEEE 39th Conference on*, pages 474–477. IEEE, 2014.

[65] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 101–114. ACM, 2016.

[66] Daniel E Lucani, Morten V Pedersen, Diego Ruano, Chres W Sørensen, Frank HP Fitzek, Janus Heide, and Olav Geil. Fulcrum network codes: A code for fluid allocation of complexity. *arXiv preprint arXiv:1404.6620*, 2014.

[67] Desmond S Lun, Muriel Médard, and Ralf Koetter. *Efficient operation of wireless packet networks using network coding*, volume 5. IWCT, 2005.

[68] Guanjun Ma, Yinlong Xu, Minghong Lin, and Ying Xuan. A content distribution system based on sparse linear network coding. In *Third Workshop on Network Coding (Netcod 2007)*, 2007.

[69] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[70] K. Menger. "zur allgemeinen kurventheorie,". *Fundamenta Mathematicae*, Vol. 10:pp. 96–115, 1927.

[71] M. Médard and A. Sprintson. *Network Coding: Fundamentals and Applications*. Academic Press. Elsevier, 2012.

[72] Felicián Németh, Ádám Stipkovits, Balázs Sonkoly, and András Gulyás. Towards smartflow: case studies on enhanced programmable forwarding in openflow switches. *ACM SIGCOMM Computer Communication Review*, 42(4):85–86, 2012.

[73] Morten V Pedersen, Janus Heide, and Frank HP Fitzek. Kodo: An open and research oriented network coding library. In *International Conference on Research in Networking*, pages 145–152. Springer, 2011.

[74] Morten V Pedersen, Janus Heide, Frank HP Fitzek, and Torben Larsen. Pictureviewer-a mobile application using network coding. In *Wireless Conference, 2009. EW 2009. European*, pages 151–156. IEEE, 2009.

[75] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[76] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[77] Fernando MV Ramos, Diego Kreutz, and Paulo Verissimo. Software-defined networks: On the road to the softwarization of networking. *Cutter IT journal*, 2015.

[78] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013.

[79] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. *AT&T Research Academic Summit, Bedminster, NJ, USA*, 2016.

[80] Danilo Silva and Frank R Kschischang. Universal weakly secure network coding. In *Networking and Information Theory, 2009. ITW 2009. IEEE Information Theory Workshop on*, pages 281–285. IEEE, 2009.

[81] Danilo Silva and Frank R Kschischang. Universal secure network coding via rank-metric codes. *IEEE Transactions on Information Theory*, 57(2):1124–1135, 2011.

[82] Dávid Szabó, Attila Csoma, Péter Megyesi, András Gulyás, and Frank HP Fitzek. Network coding as a service. *arXiv preprint arXiv:1601.03201*, 2016.

[83] Andrea Tassi, Ioannis Chatzigeorgiou, Dejan Vukobratović, and Andrew L Jones. Optimized network-coded scalable video multicasting over embms networks. In *Communications (ICC), 2015 IEEE International Conference on*, pages 3069–3075. IEEE, 2015.

[84] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM, 2012.

[85] Jun Yang, Bin Dai, Lu Lv, and Guan Xu. Coding Openflow: Enable Network Coding in SDN Networks. *International journal of Computer Networks & Communications*, 7(5):29–38, 2015.