UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Network Coding Switch

Diogo Filipe Vieira Gonçalves

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Fernando Manuel Valente Ramos
e co-orientada pela Prof.ª Doutora Muriel Médard (MIT)

2019

Acknowledgments

I would like to start by thanking my supervisor, Professor Fernando Ramos, for the opportunity to work on this project for my dissertation and for supervising it.

An enormous special thanks to Dr. Salvatore Signorello for all the patience and help throughout the entire year.

To my parents, a word of gratitude. It is only thanks to them that I was able to be where I am today.

To my big brother, thank you for being someone I can look up to.

A thank you to all my friends from college for sharing this journey with me during these years and well, because "prontos".

A thank you to all my friends for sharing so many good moments with me all these years.

Finally, my biggest thanks goes to Marta Reis, for all the support, understanding and love. I can always count on you.

This work was supported by FCT through project UPVN: User-centric Programmable Virtual Networks, ref. PTDC/CCI-INF/30340/2017, and the LASIGE Research Unit, ref. UID/CEC/00408/2019.

Resumo

O tráfego na Internet está a crescer a um ritmo elevado. A ocorrência de gargalos é, então, cada vez mais, uma ocorrência comum, resultando em atrasos no transporte de informação e em ineficiências. Isto é um problema em parte decorrente do paradigma tradicional "store and forward". Quando um pacote chega a um nó da rede, é armazenado numa fila de espera enquanto aguarda por uma decisão de encaminhamento. Quando existe tráfego elevado, as filas de pacotes crescem e os atrasos aumentam (assim como as perdas de pacotes).

O conceito de *Codificação na Rede* procura oferecer uma alternativa de paradigma. A ideia fundamental é a seguinte: à capacidade de armazenamento e encaminhamento é adicionada aos nós a capacidade de *combinar* pacotes. Com esta técnica é possível aumentar as taxas de transferência de informação, assim como a resiliência da rede.

Para se entender melhor o conceito vejamos um exemplo. Considere um nó A e um nó B que comunicam através de um ponto de acesso S, num ambiente sem fios. Vejamos as transmissões necessárias para A enviar a mensagem a a B, e para B enviar a mensagem b para A, usando o modelo tradicional:

1. A envia a para S
2. B envia b para S
3. S faz broadcast de a para os dois nós
4. S faz broadcast de b para os dois nós

Como se pode observar, foram necessárias quatro transmissões ao todo. Ao aplicarmos codificação na rede podemos poupar no número de transmissões da seguinte forma:

1. A envia a para S
2. B envia b para S
3. S combina as duas mensagens aplicando um XOR sobre elas e envia o resultado, $a \oplus b$, para A e B

No entanto, o exemplo demonstrado acima é um caso base de Linear Network Coding (LNC). Esta técnica de codificação consiste em dar a capacidade, a cada nó da rede, de gerar novos pacotes através de combinações lineares de pacotes recebidos anteriormente, multiplicando-os por coeficientes escolhidos de um dado campo finito, sendo o mais comum de tamanho 2^8. Já no exemplo anterior, em que foi utilizado uma técnica de codificação através do XOR para codificar dois pacotes, o tamanho do campo finito era de 2. Sendo este, então, um caso particular.

Porém, o LNC requere que os coeficientes utilizados nas combinações lineares sejam definidos e computados à priori por todos os nós da rede através de um algoritmo e de informação partilhada. Estamos, então, perante uma limitação desta técnica que introduz um custo. Random Linear Network Coding (RLNC), uma variante da técnica de LNC, permite ultrapassar essa limitação. Isto é possível devido à sua natureza aleatória, significando que os coeficientes empregues nas combinações lineares são gerados de forma aleatória dado um certo campo finito. Esta propriedade garante com uma dada probabilidade, desde que o campo finito tenha um tamanho suficientemente largo, de que as combinações lineares geradas sejam independentes entre si. Com o intuito de aumentar esta probabilidade, RLNC introduz ainda a capacidade de recodificar pacotes, isto é, codificar pacotes que já foram codificados por outro nó na rede.

Assim, quando o nó destinatário recebe uma quantidade suficiente de pacotes codificados que sejam linearmente independentes é possível descodificar os pacotes resolvendo as combinações lineares. Para tal, o destinatário tem de ter conhecimento dos coeficientes empregues nas combinações lineares. Então, por norma, em RLNC os coeficientes são anexados ao cabeçalho do pacote, após a codificação deste, para que os coeficientes sejam levados até ao destinatário.

Tanto a operação de codificação como de descodificação introduzem uma certa complexidade computacional proporcional ao tamanho dos dados a serem transmitidos. A técnica designada por Generation-based RLNC, permite solucionar este problema. Esta consiste em dividir grandes quantidades de dados em blocos mais pequenos, chamados **gerações**. Então, tanto a operação de codificação como a de descodificação são aplicadas por geração e não na totalidade de dados.

Existe uma grande quantidade de trabalho teórico relacionado com Network Coding e implementações ao nível da camada aplicacional. No entanto, não existe nenhum trabalho concreto cujo objetivo tenha sido desenvolver e implementar uma solução de Network Coding diretamente no plano de dados da rede. Isto resulta do facto de os switches serem hardware especializado com função única, não permitindo a codificação de pacotes.

Recentemente, no entanto, foram desenvolvidos switches programáveis, que removem esta restrição. Ao contrário dos switches tradicionais que são dispositivos fechados que seguem um conjunto de protocolos definidos pelo fabricante, estes switches permitem ao operador definir exatamente o processamento dos pacotes. Entretanto foi desenvolvida também uma linguagem de alto nível para programar estes novos switches programáveis, designada como P4.

Em suma, uma das limitações de todas as soluções de codificação em rede existentes prende-se com o facto de serem implementações em software. Esta limitação é resultado da inflexibilidade dos planos de dados em hardware (switches e routers) tradicionais, que não permitem a combinação de pacotes. Nesta dissertação começamos a atacar este problema através da exploração dos novos switches em hardware programáveis, desenhando e implementando um switch que executa Random Linear Network Coding usando a versão mais recente da linguagem de programação de switches P4 (especificamente, P4_16). A avaliação da nossa solução oferece boas perspetivas para a possibilidade de deployment em hardware destas técnicas de codificação em rede, mas apresenta também alguns dos desafios que permanecem em aberto para explorar em trabalho futuro.

Palavras-chave: Redes definidas por Software, Codificação na Rede, P4, Switches Programáveis, Random Linear Network Coding

Abstract

Network traffic is steadily increasing and, as a result, bottlenecks are becoming even more likely to occur, leading to a decrease in application performance and an increase in network inefficiencies. This is in part a consequence of a limitation of the traditional store-and-forward paradigm, used in computer networks. When a large amount of packets traverses a network, queues build up, delay increases, and overall performance decreases.

Network Coding is a new paradigm where network nodes can also *combine* packets. By coding the original packet payloads in the network it is possible to reduce the amount of information to be carried over its links, improving efficiency. In theoretical studies, Network Coding has proved to have the potential to improve the throughput, the security and the resilience of a network. However, its deployment in practice has been more difficult given the limitation of traditional hardware fixed function switches that preclude store-code-and-forward mechanisms. As a result, existing implementations are restricted to software running at end-hosts.

This work explores the design of a switch capable of performing Network Coding, something made possible recently by two factors. First, the emergence of fully programmable switches, replacing the de facto standard fixed-function switches, enables the processing of custom protocols in the data-plane. Second, the community-driven effort to define a new unified high-level language (P4) for programming the network behavior of both these latest switches and also traditional hardware and software packet processors. Hence, by leveraging the two above factors, this work aims to unleash the potential of Network Coding by running this forwarding paradigm entirely in the data-plane. A preliminary evaluation of the hereby-presented solution sheds light on some of the trade-offs involved in a practical data plane implementation of Network Coding.

Keywords: Software-Defined Networks, Network Coding, P4, Programmable Switches, Random Linear Network Coding

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The store-and-forward mechanism is the main paradigm used to forward traffic in existing computer networks. Each node in these networks receives, stores, and forwards packets to the next intermediary node until they arrive at their destination. Network Coding (NC) [1] is an alternative to this type of forwarding, where nodes are capable of more complex computation. Specifically, NC nodes do not only store and forward packets, but they can also combine several into one or more new packets, by applying a coding function (for example, by performing an Exclusive-OR).

Random Linear Network Coding (RLNC) [2] is a variant of Network Coding which brings benefits by relaxing some of the assumptions originally required to implement this network paradigm. Concretely, RLNC decentralizes code generation computation, avoiding the drawback of the original NC approaches. Moreover, RLNC grants the capability of performing recoding, which increases the probability of generating innovative information which turns useful to perform decoding. Unfortunately, the complexity of the packet processing inherent to NC precludes its deployment in conventional network data planes, given their fixed-function nature. The main focus of this work is thus to investigate the design and deployment of RLNC on state-of-the-art programmable switching technologies. More precisely, the focus is on Generation-Based RLNC, a coding technique that reduces the computational complexity of the coding and decoding operations.

This work has practical relevance by being integrated in the context of the FCT-funded User-centric Programmable Virtual Networks (UPVN) project. UPVN aims to provide a platform that gives users the ability to completely define how the elements of their virtual networks process their packets. The Network-Coding use-case aims to illustrate the ability (for evaluation purposes) of the UPVN platform to define and run complex packet processing.

This chapter serves as an introduction to the thesis. The motivation for this work is described in Sec. 1.1. Afterward, its goals and contributions are presented in Sec. 1.2. Finally, the rest of the document is outlined in Sec. 1.3.

1.1 Motivation

The theory of Network Coding was first presented in a seminal paper in 2000 [3]. While there was some progress over the past two decades, the requirements to transition from a theoretical concept to a practical deployment in a production network have not been met until very recently. Among the main barriers hindering the deployment of this forwarding paradigm is the mainstream adoption in computer networks

of fixed-function switches, that is, devices which only operate over a fixed set of standard protocols. This limitation precludes network coding in production.

The recent emergence of fully programmable switches changes the state-of-affairs by enabling the deployment of new protocols to run in various data plane targets, including in hardware. Moreover, the advent of P4 [4], the first high-level language for these packet processors, facilitates the process, by allowing a programmer to define how packets are processed in the data-plane, easing its description and increasing the portability of the so-defined network behavior.

A first attempt at this challenge was done in [5]. However, that work used the original version of the P4 language `P4_14`. Since then, the P4 language has considerably evolved and a largely improved version of the language specification, `P4_16`, is today available, deprecating `P4_14`. Today, `P4_16` is the stable, and expected more future-proof version of the language, and the mainstream focus of the research community, thus becoming the reference for any present and future work with the language.

This thesis improves upon [5], by using `P4_16`, and, even more importantly, by considering and exploring multiple design choices, alongside a more sound evaluation.

1.2 Goals & Contributions

This work explores the potential of the latest generation of programmable switching chips to run network coding operations at line-rate in the data-plane. For that purpose, its main goal is to design, implement, and evaluate a network coding switch written in the high-level language P4. More precisely, this work investigates the feasibility of RLNC [6] in real switches.

In addition to the design and implementation of the first RLNC data plane developed in *P4_16*, we also made the following contributions:

- the definition of a new *packet format*. Since there exists no standard packet format for network coding solutions we followed the general guidelines from the IETF [7].
- the capability of the data plane to employ both coding and recoding operations, depending on the original packet being uncoded or coded, respectively.
- the exploration of different coding configurations and diverse finite field arithmetic algorithms, to explore the trade-offs.
- the evaluation of not only the functional correctness of the solution (as in [5]) but also its performance, by studying the impact of different coding configurations, and by analyzing the cost of the recoding operation.

As explained above our implementation of a network coding switch targets the latest version of the P4 language, namely `P4_16`. As the first target platform for evaluation, the P4 program was tested on the reference platform for language development, the P4 software switch [8]. The basic functionality of our solution is thus validated in this stand-alone software switch. The same program was then stress tested, on the same software switch, by use of a traffic generator. Metrics regarding the throughput, CPU utilization, and packet loss were collected and analysed. This evaluation seeks to shed light on some of the trade-offs involved in a practical data plane implementation of NC. The reader may be wondering why a solution

motivated to be deployed in hardware was evaluated only in a software switch, and what the implications are. It is therefore important to stress that a P4 implementation should, in principle, run in any target data plane that supports the language (excluding platform dependent extern functions that may need to be ported). This includes hardware switches (e.g., Barefoot Tofino), smartNICs (e.g., FPGA-based), and software switches (e.g., Open vSwitch). Given the limited time of a masters project, we started with an evaluation on the reference P4 switch in hardware, but the next step, as future work, is to run it in a Barefoot Tofino and in a NetFPGA available in our laboratory.

The result of this work is made available in our working group's repository[1]. Besides our RLNC implementation, it also includes tools to automatically generate P4 code based on the coding parameters defined by the user and sender/decoder/measurement applications to generate traffic and validate/evaluate the solution.

Finally, a paper with the work we present here was accepted at the EuroP4 2019 Workshop[2] [9].

1.3 Structure of the document

The remaining of this document is organized in the following way:

- Chapter 2 - details the background and related work. The first sections provide the background for network coding and programmable networks. In the last section, we detail closely related work.
- Chapter 3 - presents an in-depth description of the proposed solution, as well as the reasoning behind our design choices. We start by exposing the principles and assumptions and the general architecture of our solution as well as the packet workflow. Then, we present the packet format to be used with an RLNC network protocol. After that, we give a detailed explanation of the buffering module, one of the crucial components of our architecture. Finally, we explain the algorithms employed to perform finite field multiplication.
- Chapter 4 - covers the most relevant technical details related to the implementation of the proposed solutions. Such details include the definition of the headers and parser, how we implemented a custom buffering mechanism, how we achieved packet replication, as well as the implementation of the finite field multiplication algorithms.
- Chapter 5 - includes an evaluation of our solution. We start by presenting the developed tools for evaluation purposes and the testing environment, and then we present and discuss the results of our practical experiments.
- Chapter 6 - summarizes the thesis and discusses future work.

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

[2] In this workshop 6 full papers were accepted out of 18 submissions.

Chapter 2

Related Work

This chapter is organized as follows. An introduction to Network Coding that illustrates the main principles of its most relevant variant, is given in Sec. 2.1. Then, Software Defined Networks are introduced, followed by a description of the P4 language in Sec. 2.2. As mentioned in the previous chapter, this work is also intended to be a use-case for the virtualization platform developed within the context of the UPVN project. Therefore, we briefly explain the state-of-the-art of network virtualization. Finally, Sec 2.3 presents a closely related work, namely a thesis that was a first attempt to implement network coding on programmable switches.

2.1 Network Coding

Traditionally, in computer networks packets are simply forwarded from sources to destinations along distributively precomputed routing paths. This behavior is referred to as the store-and-forward paradigm in computer networks. Network Coding [1], instead, empowers nodes with additional computing capabilities so those can transmit functions of the packets they receive. In particular, practical Network Coding schemes usually leverage linear algebra over packets payload to code information for several packets.

Network coding has been proven to provide many advantages. Maximizing throughput of a network is the most evident one [10], especially in multicast scenarios. Better tolerance for packet losses [11] and network failures [12], and an improvement in security [13], are some of the other benefits that network coding may bring.

Network Coding has had its fair share of applications in a variety of environments while bringing many of its benefits to them. It has been shown that it can improve the throughput in wireless networks in COPE [14]. COPE codes packets without making assumptions about topology, identifies coding opportunities and forwards multiple packets in a single transmission. The design of COPE capitalizes on two key factors. First, it relaxes the point-to-point communication assumption and fully embraces the broadcast nature of a wireless channel. Second, it employs network coding. Therefore, it combines packets together before forwarding them.

In the field of on demand streaming services network coding has also shown its benefits. The UUSee [15] is a service that provides on-demand streaming. It uses a protocol specifically designed from the ground up to incorporate network coding. It scales up to thousands of on-demand video channels and millions of users, without the penalty of excessive server bandwidth costs.

PictureViewer [16] is an application that shows the practicality of NC on mobile phones. PictureViewer

uses different coding algorithms including using RLNC (explained later) to send information, and a systematic strategy where it sends both coded and uncoded information. This application allows users to broadcast images located on their phones to a number of receiving devices. The main idea is that users share content over short range technologies such as WiFi. Instead of uploading the content to social networks, the content can be conveyed directly to mobile phones in the vicinity.

Kodo [17] is an erasure coding library with a special focus on network coding algorithms and codecs. It is written in the C++ language and designed to ensure performance, testing, flexibility, and extensibility. The Kodo library is meant to be used for commercial applications and research on the implementation of Network Coding.

These are just some of the many applications that employ network coding. The study of these past works show the increasing prevalence of network coding in different fields and the growing interest in this new network paradigm.

2.1.1 The Butterfly Network Case

A widely-used example to showcase the advantages of Network Coding is the multicast throughput in a butterfly network, which is the directed graph illustrated in Fig. 2.1. In the butterfly network example, there is a single source node S, two destination nodes T and U and some other intermediate nodes including B and C. Each directed edge in this network represents an error-free packet link, capable of delivering only a single packet per time unit.

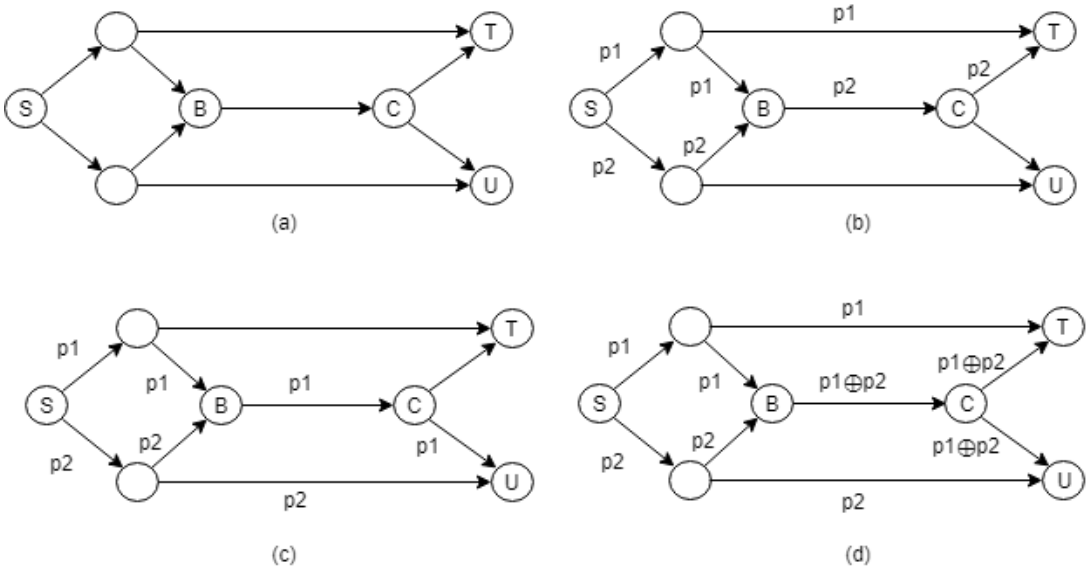


Figure 2.1: **(a)** The butterfly network **(b)**, Traditional routing solution to T, **(c)** Traditional routing solution to U, **(d)** Network coding solution

S produces two packets, $p1$ and $p2$, with destination T and U. Using the traditional routing solution (illustrated in (b) and (c) of Fig. 2.1), the link between B and C represents a bottleneck. In fact, packets $p1$ and $p2$ cannot travel on the same link at the same time. Thus, a queue is formed in B and two-time units are needed for both packets to finally arrive at their respective destinations, T and U. This is the best multicast throughput that can be achieved through any routing solution [1] on this network.

A network coding solution may easily improve the multicast throughput in the butterfly network

example (see (d) of Fig. 2.1). In fact, $p1$ and $p2$ can be combined at B by applying a logical exclusive-OR (XOR) on both packets and transmitted as a single packet consuming only a one-time unit on the link B-C. Destination T now receives $p1$ and $p1 \oplus p2$. T can decode and recover $p2$ from $p1 \oplus (p1 \oplus p2)$. Similarly, destination U receives $p2$ and $(p1 \oplus p2)$ and can also decode and recover $p1$ from $p2 \oplus (p1 \oplus p2)$.

This network solution is the optimal one to achieve the highest multicast throughput. This result is proven by the the "max-flow min-cut" theorem [18].

2.1.2 Linear Network Coding

In Linear Network Coding (LNC), encoding consists in each node generating new packets which are linear combinations of earlier received packets, multiplying them by coefficients chosen from a finite field, typically of size 2^8. Each node generates an output message p from the linear combination of the received messages m_i by using the equation:

$$p = m_1 \times c_1 + m_2 \times c_2 + ... + m_i \times c_i$$

Where the values c_i are coefficients selected from the finite field F. The XOR encoding scheme used in the butterfly network example before is, in fact, a particular case of this more general approach, where the size of the finite field is 2.

2.1.3 Random Linear Network Coding

RNLC is a variant of LNC where the coefficients used for the linear combinations of the packets are selected randomly by each node within a certain finite field[1]. Conversely, LNC requires these coefficients to be computed a priori by all the nodes using the same algorithm and some shared information. Another advantage of RLNC is that, nodes may perform recoding (i.e., coding packets that have already been coded by other nodes). Recoding increases the likelihood of innovative[2] transmissions to the receiver.

To illustrate RLNC through an example, consider a network composed of two senders, S1 and S2, several intermediate nodes and a destination D. S1 and S2 send respectively packets p_1 and p_2 to D. At any intermediate node, these packets can be combined in a single packet by performing linear combinations of their data using randomly generated constant coefficients r_i. For example, an intermediate node could combine two packets and produce two different encoded packets, named p'_1 and p'_2, by using the following computation scheme:

- p'_1 = $r_1 \times p_1$'s data + $r_2 \times p_2$'s data
- p'_2 = $r_3 \times p_1$'s data + $r_4 \times p_2$'s data

After encoding the packets as above, the node sends them towards the destination. At the reception of the encoded packets, D can retrieve the original packets through a process of decoding. The decoding consists in solving the linear equations which, however, requires knowing the coefficients r_i. Therefore, coefficients need to be sent along with packets (e.g., in the packet header).

[1] Note that all the arithmetic operations in LNC are performed inside a pre-established finite field.

[2] A transmission is said *innovative* when it contains new information allowing a node to progress with the decoding process.

2.1.4 Generation-Based RLNC

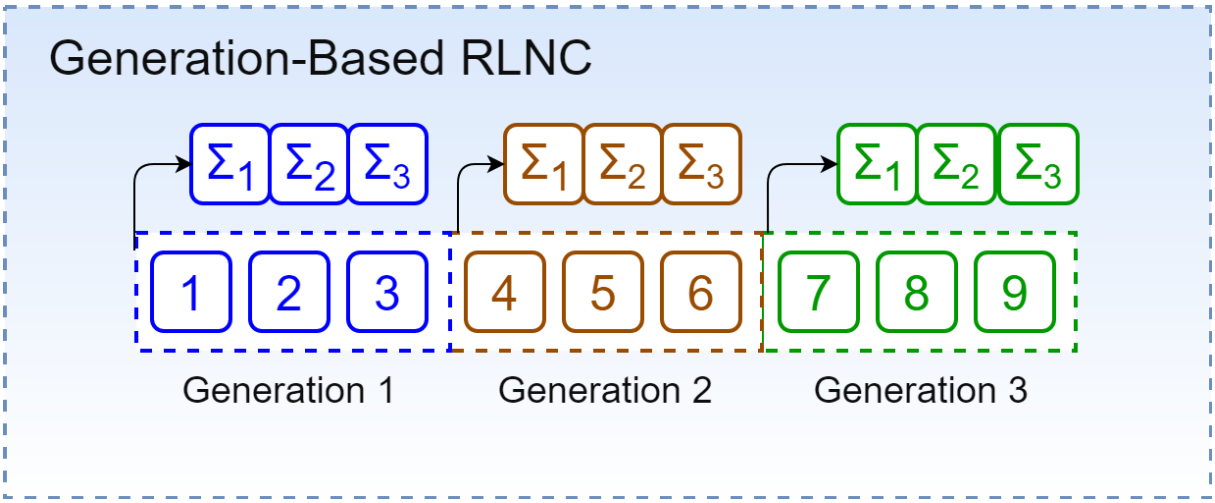


Figure 2.2: A generation based approach to RLNC

Encoding and decoding operations in RLNC introduce a certain computational complexity which is proportional to the size of the data to transmit. To tackle this issue, there exist RLNC techniques which split large data into smaller equal-sized chunks, called *generations*. These techniques are commonly called generation-based. In RLNC, a generation consists of *symbols*, which are vectors of elements representing the data. Coded symbols are coupled with *coding vectors* containing the coefficients r_i for a symbol in the generation. By having the coefficients and the encoded symbols, a receiver can solve the linear equations and obtain the original source symbols.

This block-by-block (generations) approach reduces the computation complexity of the encoding and decoding operations (Figure 2.2). Generation-Based RLNC allows for an efficient, practical RLNC encoding and decoding across diverse platforms and network settings. Nevertheless, RNLC has two main pitfalls. It introduces delay and packet overhead. In Generation-Based RLNC a node has to wait until it has received all the symbols of a generation to start the encoding operation. Only then will the network node start the forwarding process. Nevertheless, there are different strategies that try to reduce the delay introduced. Besides the one previously described, designated by Standard RLNC, some approaches have been proposed:

- **Sparse RLNC with uniform density:** This schema is akin to the standard RLNC, yet some symbols are excluded with a given probability during encoding. This is useful when the generation size is large, since the density of the code is reduced without having a negative impact and the decoding performance is greatly increased.

- **Sparse RLNC with fixed density:** A fixed number of symbols are combined at random within the same generation. This technique turns very useful when feedback is available from a receiver, which decodes the packets, as the encoding/recoding processes can be adjusted accordingly.

- **Seed-based RLNC:** This schema transmits over the network a small random seed used to generate the coding vectors instead of sending the full vectors. This technique reduces the overhead but

it notably increases the difficulty of recoding. This type of coding is used when recoding is not necessary or very rarely used.

- **On-the-fly or Sliding-Window RLNC:** This technique does not require all the symbols to be available before starting with the encoding/decoding operation. Coding is performed across a dynamic set of symbols by means of variable-sized sliding windows. All uncoded symbols within the window range are considered for creating coded symbols. This sliding approach introduces less delay compared to the Generation-Based RLNC schemes. However, it relies on assumptions, such as infinite or dynamically-sized windows, which make its implementation difficult.

2.1.5 A Primer on Galois Fields

In RLNC, linear combinations of packet's elements are strictly performed over finite fields (also called Galois Fields). A Galois Field (GF) is a complex mathematical construct that can not be covered here in its entirety. However, a basic explanation is given so that the reader can have a basic understanding of its concepts.

A Galois field is a field containing a finite number of elements meeting certain arithmetic properties. For the $GF(2^n)$ field, addition, subtraction, multiplication and division operations are defined over the numbers $0, 1, ..., 2^n - 1$ that:

- All of the operations are closed. That is, if a and b are elements of the field then the resulting value of $(a + b)$, $(a - b)$, $(a * b)$ and (a/b) is also an element of the field
- basics properties (e.g. commutative, associative, etc.) of addition, subtraction, multiplication and division are verified.

Finite fields of order 2^n are called binary fields. They are of special interest because they are particularly efficient for implementation in hardware, or on a binary computer. The elements of $GF(2^n)$ can be represented in one of four ways:

- As a decimal number between 0 and $2^n - 1$.
- As a hexadecimal number between 0 and $2^n - 1$.
- As a binary number with n digits.
- As a polynomial of degree $n - 1$, whose coefficients are binary.

Arithmetic operations in a Galois Field do not follow the same rules of the conventional mathematics. Instead, it has its own specific rules. The operations that are given most attention are addition and multiplication in a Galois field over 2^n, due to their importance in encoding packets. A more detailed description is provided below.

In $GF(2^n)$, addition is trivial. To add or subtract two elements of a finite field we simple perform a XOR operation over the two elements. For example, 25 + 10 is equal to 19:

$$25 + 10 = (x^4 + x^3 + 1) + (x^3 + x) = x^4 + x + 1 = 19$$

The equivalent, in binary form, is the following:

$$25 + 10 = 11001 + 1010 = 10011 = 19$$

Multiplication is far more complex [19]. To multiply two numbers a and b, the first step is to multiply their polynomials. If the result of the multiplication has a degree less than n, then the operation is considered completed. For the following examples, consider a $GF(2^4)$. To multiply numbers 2 and 5:

$$2 \cdot 5 = (x)(x^2 + 1) = x^3 + x = 10.$$

Consider now an example where the multiplication results in a number whose polynomial representation has a higher degree that n:

$$\begin{aligned} 10 \cdot 13 &= (x^3 + x)(x^3 + x^2 + 1) \\ &= x^3(x^3 + x^2 + 1) + x(x^3 + x^2 + 1) \\ &= (x^6 + x^5 + x^3) + (x^4 + x^3 + x) \\ &= x^6 + x^5 + x^4 + x. \\ &= 114 \end{aligned}$$

In this case, the product p needs to be reduced to an equivalent polynomial p' whose degree must be less than n using a special polynomial of degree n denominated the *irreducible polynomial*, R. We will not dwell on the construction of the irreducible polynomial. Consider, for the sake of this example, the irreducible polynomial to be the polynomial $x^4 + x + 1$:

$$\begin{aligned} p' &= p \bmod R \\ &= (x^6 + x^5 + x^4 + x) \bmod (x^4 + x + 1) \\ &= 114 \bmod 19 \\ &= 11 \end{aligned}$$

The above information only summarizes the definition of a Galois field and of some of the arithmetic possible on those fields. A more comprehensive study can be found in [20].

2.2 Programmable Networks

In this section, we study the state of the art on Software-Defined Networks (SDN), how programmable data planes came to be and its current status, including an explanation of the intricacies of the P4 language. Next, we give an overview of network virtualization, the field of the project (UPVN) in which this work is inserted. Finally, we end this section by detailing a previous attempt on implementing network coding on a switch data plane.

2.2.1 Software-Defined Networking

Traditional computer networks are divided into three planes of functionality:

- Data plane: Consists of the devices forwarding the data.

- Control Plane: Populates the tables of the devices in the data plane.
- Management Plane: Monitors and configures the control plane.

Despite traditional IP networks being widely used and adopted, they remain very complex and hard to manage [21]. For example, network operators often need to configure each individual network device manually or with the use of ad-hoc hard-to-reuse scripts. Yet, those networks need to dynamically adapt to traffic demands and endure faults, which, in the end, results in a very inefficient system. Besides, the introduction and deployment of a new protocol, when possible, is a several-years long process, so network innovation is slow,

Software-defined Networking (SDN) is a new networking paradigm, aiming to solve the shortcomings of the traditional computer networks. One key characteristic of the SDN is that it breaks the vertical integration in today's network infrastructure by decoupling the control plane and the data plane, giving greater flexibility and space to innovate. By decoupling the planes, switches become simple forwarding devices and the control plane can become logically centralized. This makes it easier to observe and reason about the network behavior while also enhancing network-wide visibility and having more direct control over the network traffic [21].

Key concepts of a Software-Defined Network are:

- Control and Data plane decoupled: Switches are now simple forwarding devices since control functionality is removed from these.
- Forwards decisions based on flows: A flow is a sequence of packets from a source to a destination. By using this abstraction, the unification of the behavior of different types of network devices is enabled, increasing flexibility.
- Control logic now runs in an external entity: The control plane is moved to a commodity server: the SDN controller. This way, the programming of the forwarding devices in the network is eased.
- Programmable Network: The network can be programmed by using software applications running on top of the controller.

The typical architecture of an SDN is depicted in Fig. 2.3. From the bottom to the top, the lowest layer represents the data plane where all the forwarding devices operate. In the middle layer, there is the control plane, where the control logic resides. At the top, there is the application layer, where the network applications run. Two interfaces, namely the northbound and southbound API, interconnect the middle layer respectively to the top and bottom ones. The northbound interface is not standardized yet, while the de facto standard for the southbound interface is the OpenFlow (OF) protocol [22].

2.2.2 Data plane programming in P4

The SDN decoupling of the control plane from the forwarding plane, with one control plane controlling several forwarding devices, enabled for the control of forwarding devices to be programmed in different manners [4]. OpenFlow, a common, open, vendor-agnostic interface enabled a control plane to control forwarding devices from different hardware and software vendors. The SDN vision paired with the OF protocol has given unprecedented ability to program the network.

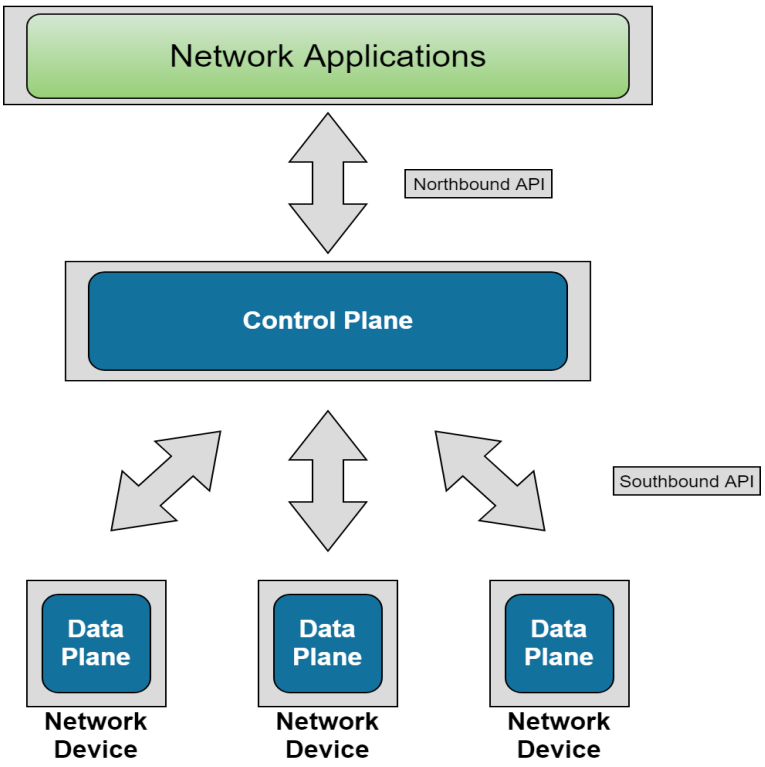


Figure 2.3: Software Defined Network Architecture

However, OF is still very dependent on both the underlying available network equipment and the standard protocols used in IP-based networks. The initial OF specification grew with time for switches to expose more of their capabilities to controllers. This growth showed no signs of stopping. Meanwhile, new chip designs demonstrated that programmable parser and match-action logic could be achieved in high-speed switching chips [23]. However, each chip may still expose its own low-level custom programming interface. Instead of extending OF to program the latest and the old switches, a common language to program every kind of switch in a unified and highly portable way was proposed: The Programming Protocol-independent Packet Processors (P4) language, initially proposed in [4].

P4 is a high-level programming language designed to program the data plane of P4-compliant network forwarding devices, called P4 targets. P4 was designed with three main goals in mind:

- Target independence: A P4 program is meant to be independent of a specific target and is rather capable of running on several compliant devices. This is achieved through target-specific compilers mapping a unique P4 program down to different target platforms. So, the language does not require any knowledge about a specific platform to write programs.
- Protocol independence: P4 is designed to be completely oblivious to any specific existing protocols and is generic enough to express the most varied network behaviors.
- Reconfigurability: P4 should allow a programmer to change the packet processing logic even after a program is deployed.

P4 is designed to specify the data plane functionality of a target, while it does not describe the control-plane logic. The core abstractions used in P4 are:

- *Headers*: Describe the format, the set, and size of the fields of each header.
- *Parser*: Describe the sequences of headers in received packets. It is meant to identify the headers and fields to extract.
- *Match-Action-Tables*: Perform the following sequence of operations:
  - Construct lookup keys,
  - Select an action based on the matching key,
  - Executes the selected action.
- *Action*: Describes operations to manipulate packets and packet fields.
- *Control Flow*: Specifies the packet-processing logic on a target.
- *User-defined metadata and Intrinsic metadata*: data structures that can be associated with each packet. The first is defined by the programmer while the second one is provided by the target.

After a packet arrives, it is parsed. The parser code identifies and extracts fields from the packet according to the header definitions provided in the running P4 program. The extracted fields are then passed to the processing pipelines where various match+action tables are executed. In the pipelines, several operations on a packet, e.g., header fields manipulation, can be executed and finally forwarding decisions are taken.

As of today, two versions of P4 are available, `P4_14`, the oldest version, and `P4_16`, the latest version [24]. `P4_16` has introduced some significant changes to the syntax and the semantics of the language making it backward incompatible with the previous version. A core objective of `P4_16` is to provide a stable language definition to ensure that current programs written in `P4_16` will remain syntactically correct and behave identically with future versions of the language.

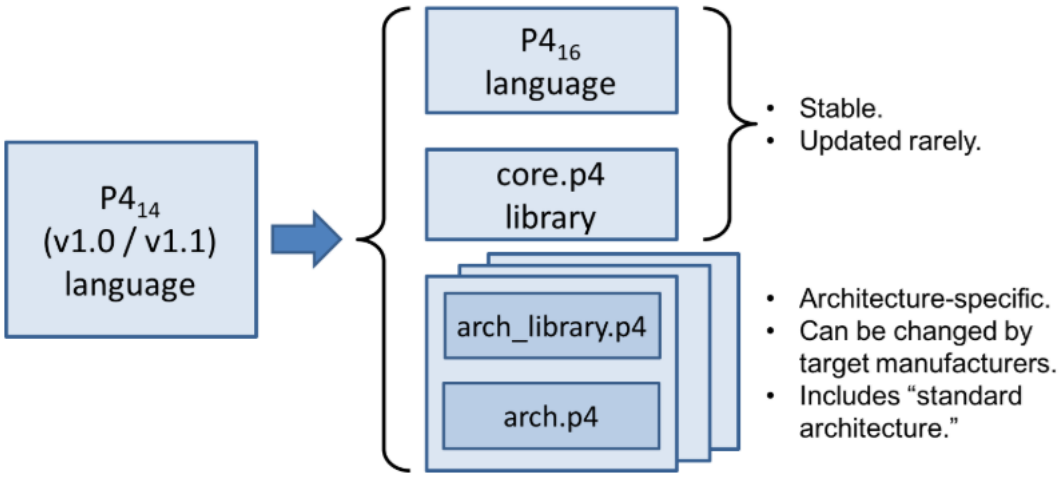


Figure 2.4: Evolution of the P4 language

Among the major changes in `P4_16`, there is the language and architecture separation. That is, there exist a set of core language constructs, which are supposed to be updated rarely, and then there are

libraries, which are architecture-specific descriptions varying over targets. The P4 architectures libraries are meant to be provided by the target manufacturers.

Another core addition of `P4_16` is the extern objects to support target-specific functions. Externs are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is not programmable in P4.

2.2.3 user-centric Programmable Virtual Networks

As this work is integrated in the FCT-funded uPVN project, we introduce it here, to contextualise. We start by briefly explaining the context: network virtualization.

Network virtualization indicates the coexistence of multiple virtual networks on the same physical substrate [25]. It is a technique that allows multiple heterogeneous virtual networks to run on a shared infrastructure while isolated from each other.

Complete network virtualization requires the decoupling of the logical service from its physical realization. Traditional networking techniques were not designed to achieve this. Fortunately, the advent of the SDN paradigm helped overcome this problem. In fact, recent works in network virtualization, e.g., FlowVisor [26], take advantage of the SDN model. In a nutshell, FlowVisor slices the network hardware by introducing a software slicing layer that sits between the control and data plane. However, full virtualization can not be achieved just by means of this slicing technique.

Production level platforms that offer this form of network virtualization have been developed in recent years. They give users the flexibility to run virtual networks with customized topologies and addressing schemes. For example, the seminal work NVP [27] leverages SDN principles, specifically its logically centralized controller, for configuring the virtual switches that compose a virtualized network infrastructure. Other examples include Google Andromeda [28] and Microsoft AccelNet [29]. These platforms have changed the status quo, but they are not without limitations. Specifically, the network services provided still stick to conventional networking.

The user-centric Programmable Virtual Networks (uPVN) project aims to address the limitations of current network virtualization solutions by designing a platform to create and manage user-centric virtual networks that are fully programmable. The UPVN platform aims to build services upon a substrate made of fully programmable network elements, which brings higher levels of customizability to the domain of virtual networks. The network program developed in this thesis serves as one of the main use cases of this project.

2.3 Network Coding with Programmable Switches

As far as we know, the master's thesis in [5] was the first attempt to implement Network Coding on programmable data-planes through the use of the P4 language. That work implements two types of coding schemes, XOR and RLNC, in P4. The two P4 programs are respectively named P4-XOR Switch and P4-RLNC Switch. Both programs follow the same architecture, illustrated in Fig. 2.5.

The Data Plane pipeline of the P4 programs consists of three main modules:

- **Buffering Module**, which stores arriving packets for future encoding opportunities.

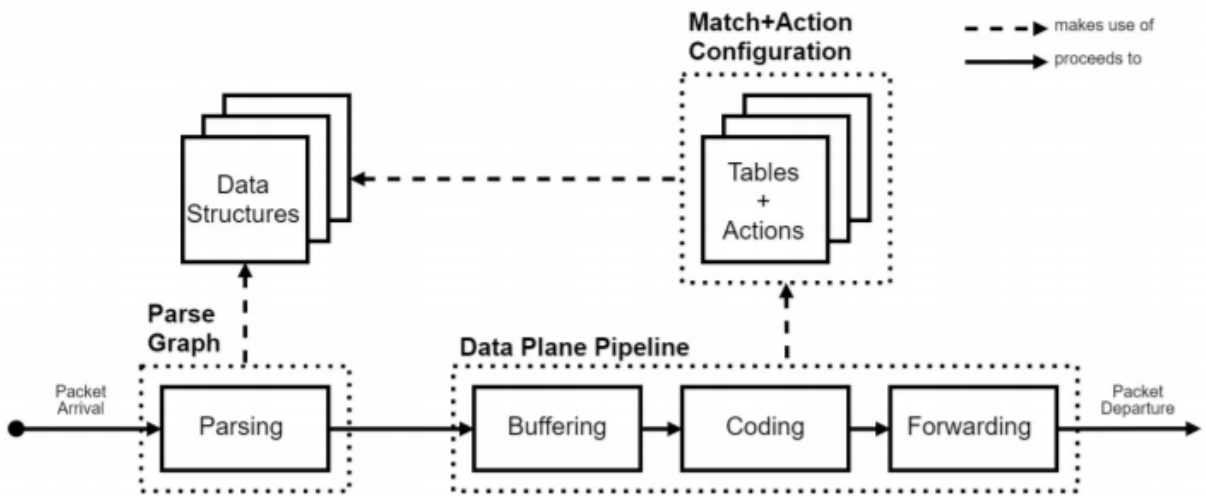


Figure 2.5: Representation of the architecture of the P4-XOR and the P4-RLNC Switch taken from [5].

- **Coding Module**, which codes or recodes packets previously stored in buffers.
- **Forwarding Module**, which sends the packet to the next node in the network.

Both P4 programs use MPLS-like headers to tag and identify packets to be coded together. For this purpose, those headers carry information such as the packet's flow[3] and a sequence number. In Network Coding, the term intraflow identifies a solution where the merged packets all belong to the same flow.

The **P4-XOR switch** program basically XORs the payload of packets tagged with the same identifier. This solution was successfully validated by measuring the performance improvement achieved by this technique in a butterfly network scenario (previously illustrated in Sec. 2.1.1).

The **P4-RLNC program** implements a single intraflow RLNC schema. In this program, MPLS-like headers also carry the coefficients used in the linear combination, needed to perform the RLNC coding/recoding/decoding operations.

In this solution, a packet's payload is designated as a *symbol* and further divided into *elements*, which are values within a GF(2^8) Galois/Finite Field. *Elements* are the basic unit used during the encoding/recoding/decoding process. The simple test scenario, experimented in [5], for this program is illustrated in Fig. 2.6, and consisted of two hosts, and one switch running the P4-RNLC program.

The source node *h1* generates, codes and sends traffic to the switch *s1*. The switch *s1* performs recoding over the traffic received and sends it to the host *h2*. The host *h2* finally decodes the coded packets to retrieve the original symbols.

While it was a good starting point, this work is currently lacking in certain aspects. First, the choice to use simple MPLS headers was an inefficient work-around. Instead, in this thesis we propose the design and implement, from the ground up, a custom RLNC packet header format following IETF recommendations. Second, the intermediate nodes only possess the capability to recode packets, this being an operation which is more complex than just coding, it would make sense that a node that can recode should also be able to code. Third, the evaluation of the P4-RLNC solution was lacking with an exploration of the impact

[3] In this example, a flow identifies a stream of data coming from a single host.

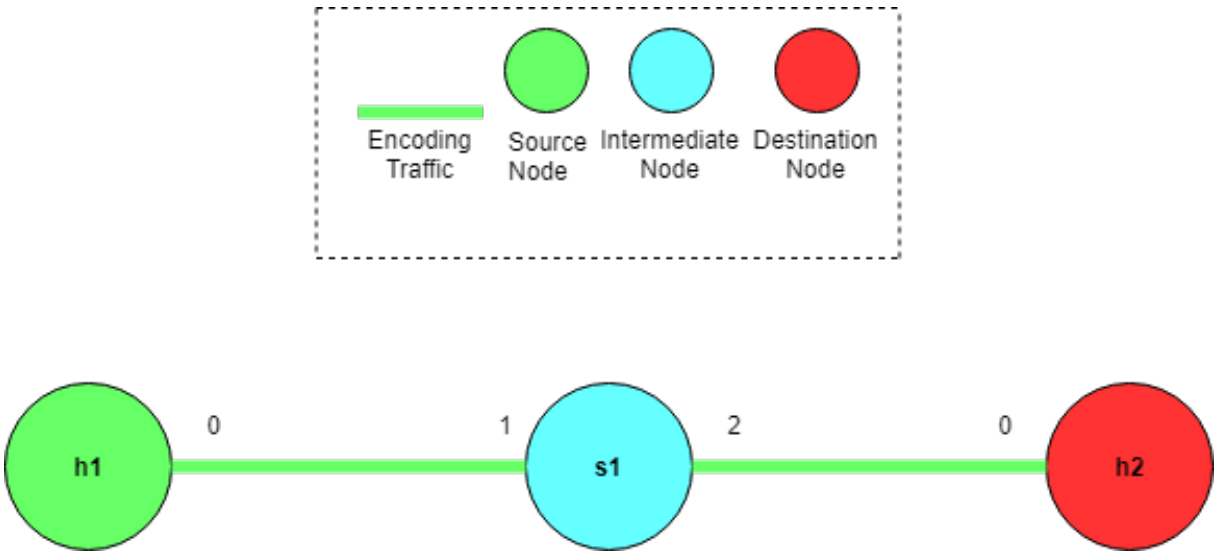


Figure 2.6: The topology used to validate the P4-RLNC switch program.

of different coding parameters (e.g. generation size) can have on the performance of the switch. Finally, this implementation is in the deprecated version of P4, P4_14.

2.4 Summary

In this chapter, the fundamentals of Network Coding were explored in Sec. 2.1. This section started with the classic butterfly network example to explain the basic and proceeded to detail how Random Linear Network Coding emerged and how a generation-based approach followed to reduce the computational complexity of the operations. It followed with a brief explanation of Galois Fields, in which RLNC is performed. In Sec. 2.2, we went through the context and motivation for the emergence of programmable networks. Finally, the chapter closes with Sec. 2.3, with an overview of the main related work and its limitations.

In the subsequent chapters we propose the design and describe an implementation for a programmable network switch capable of performing Random Linear Network Coding.

Chapter 3

On the design of a Network Coding Switch

In this chapter, we present the design of our Network Coding Switch entirely from an abstract point of view. Only in the following chapter will we dive further into the specific implementation details. Our design is driven by the limited expressiveness of the current P4 language specification, meant to describe packet processing behaviors in the data-path of high-speed forwarding devices.

Our Network Coding (NC) Switch design implements a Generation-Based RLNC protocol where encoding and decoding operations are performed over generations. More specifically, the NC switch implements a Standard RLNC behavior, where it waits for all the symbols belonging to the same generation to be received before starting coding.

This chapter is organized in the following way. The general architecture of the solution and the packet workflow is described in Sec. 3.1. Afterward, the protocol header format used for RLNC is introduced in Sec. 3.2. A more detailed description of the design of the buffering mechanism is given in Sec. 3.3, followed by an explanation on the design choices for the finite field arithmetic operations in Sec. 3.4.

3.1 Architecture and Packet Workflow

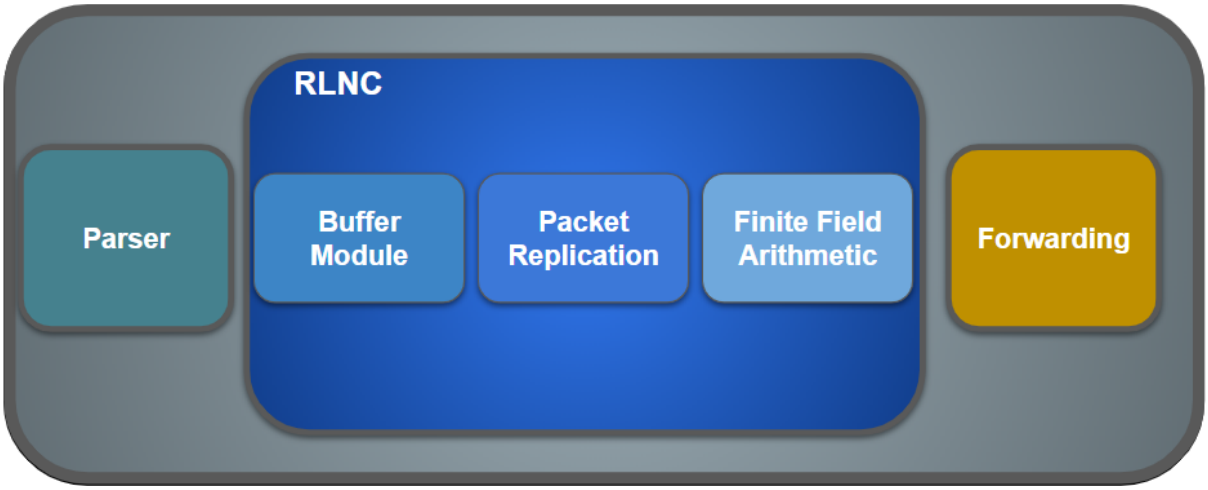


Figure 3.1: Abstract representation of the architecture of our RLNC switch.

The scheme in Figure 3.1 abstracts the main building blocks of our network coding switch. The parser validates and extracts the required information according to the packet format we specify. The

RLNC block works on the extracted parsed packet representation. The RLNC block consists of three modules. The Buffer Module handles the buffering of incoming packets, storing the necessary information for further coding operations. The Packet Replication block generates packet copies from an incoming packet. The Finite Field Operations block performs the required arithmetic over a finite field. Finally, the Forwarding block makes forwarding decisions.

Our network coding switch processes incoming packets according to the following workflow. The first thing that the packet is subjected to is the parsing stage, where we extract the information contained in the headers of the packet. The second stage is of our own design: a buffering stage where the switch stores the contents of the packet. More precisely, a stage where we buffer the symbols that compose the packet, and their coefficients. The symbols and coefficients are needed in a later stage when the switch has the opportunity to initiate the coding process.

The following stage is **Replication**. In Generation-based RLNC, a destination node must receive a number of packets equal or greater than a generation size before starting the decoding process. Therefore, for the decoding operation to be correctly performed, the coding node may need to generate a variable number of coded packets, depending on the network conditions and on the linear independence of the produced combinations.

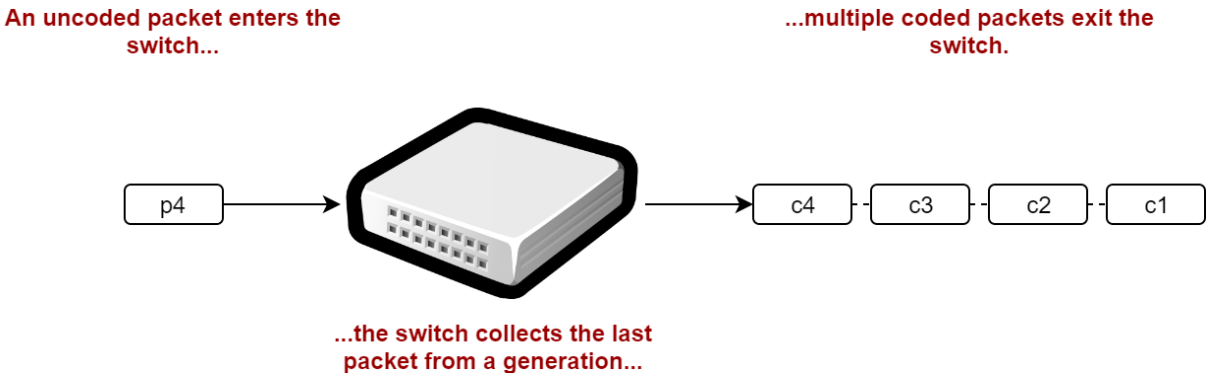


Figure 3.2: Required behavior for the Network Coding Switch

To encode a generation, our RLNC switch thus creates multiple packets, each containing different linear combinations of the buffered symbols for that generation. This operation leverages packet replication primitives (more precisely, a multicast mechanism) available on the programmable target switch.

Our RLNC switch starts the coding process only when it has buffered a *sufficient* number of packets for a certain generation. The reception of the last packet of a certain generation triggers the coding process in the switch. Then, the switch generates multiple replicas of that same packet meant to carry different linear combinations of the coded generation (see Figure 3.2).

Each packet replica then enters the **Coding** stage. As of today, there are several variants of RLNC. The majority of those coding schemes are only implemented at the application level where there are less computational and memory constraints. However, since our work targets the deployment of the protocol in a switch data plane, we have opted for the Standard RLNC technique. In Standard RLNC, all the symbols of a generation must be available to start the coding process.

An intermediate switch can code or recode depending on whether the incoming packet is uncoded or coded, respectively. In any case, the coding or recoding process only starts when the buffer space allotted

for a given generation reaches its full capacity. That means all packets belonging to a generation have been received, which is a necessary condition for Standard RLNC to be applied.

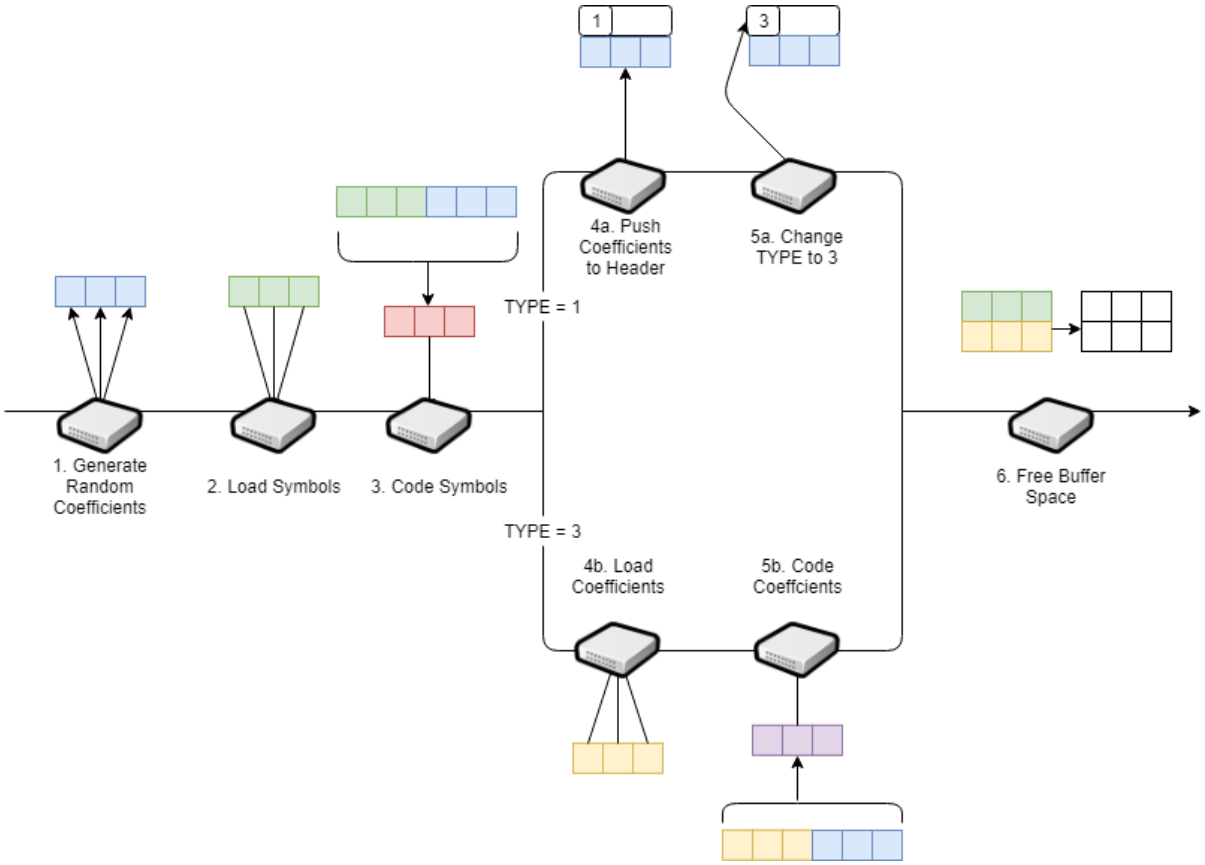


Figure 3.3: The Coding Process

The coding process is illustrated in Figure 3.3. The first step is to generate the coefficients. Since, in RLNC all the coefficients used in the linear combinations are randomly picked, a Random Number Generator is required. Its only function is to generate the required random coefficients that belong to the chosen Galois Field.

The following step is to get all the symbols belonging to the same generation that are currently stored in the switch. Then, the switch codes the symbols using the generated random coefficients.

The coding process can branch into different stages depending on whether the packet is being coded or recoded. We use the header field *TYPE* to indicate if the packet is uncoded or coded. If it has a value of 1, the packet is uncoded. Thus, the switch employs the coding operation. When it completes the process, the switch appends the generated random coefficients to the packet header and changes its *TYPE* header field to 3 (coded packet).

If the *TYPE* of the packet received is 3, a coded packet is to be processed. In this case, the switch applies recoding. The main difference of recoding from coding is that, instead of only coding over the symbols, it is also necessary to update the coding coefficients.

After the coding/recoding process for a generation is complete, the corresponding buffer space is released, and is now ready to be used for storing other generations.

Finally, the packet reaches the **Forwarding** stage. The forwarding of a packet can have two different outcomes, depending on the state of the switch. When the switch has finished collecting all the symbols

belonging to a single generation, the switch forwards the packet(s). Otherwise, if the buffer has yet to gather all the symbols related to a generation, after the packets' contents have been buffered, the switch drops the packet.

3.2 Packet Format

Random Linear Network Coding is not yet a standardized internet protocol. As such, there is no standard header format to adhere to when implementing RLNC protocols. This gap prompted the challenge of designing our own header format.

The custom packet header format designed in the context of this thesis follows the symbol representation documented in the IETF draft [7]. The decision to follow this representation stems from the desire to base our header format on a specification that may potentially become a standard in the near future.

The packet header format consists of an outer header and an inner header. The former specifies generation-based coding information. The latter describes the symbol representation for RLNC.

The outer header fields are the following:

- **Generation ID:** identifies the number of the generation.
- **Generation Size:** indicates the number of symbols in a generation.
- **Field Size:** specifies the size of the finite field, and consequently the size of each symbol

Following the outer header is the inner header, which contains the following fields:

- **TYPE**: indicates the type of the symbols in the packet. It equals 1 for systematic (i.e., uncoded) symbols, 2 for coded ones with a seed, and 3 for either coded or recoded symbols that include coefficients.
- **Symbols:** the number of symbols included in the packet.
- **Encoder Rank:** represents different things depending on the type of the packet. If *TYPE* is 1, then the encoder rank is the index of the first data symbol. If *TYPE* is either 2 or 3, the encoder rank represents the number of data symbols over which coding was performed, for all coded symbols in the packet.
- **Seed:** used to generate the coding coefficient vector(s) using a pseudo-random number generator. This field is only present when the *TYPE* field is 2.
- **Coefficient Vector:** represents a list of the coding vectors used to generate the coded symbols. This field is only present when the *TYPE* field is 3.

The above packet format can produce different headers, depending on whether the carried symbols are systematic, coded or recoded. The current version of our NC switch design does not support the seed format, that is, it is not able to generate coefficients from seed numbers carried in packets. The two supported formats are illustrated in Fig. 3.4 and 3.5.

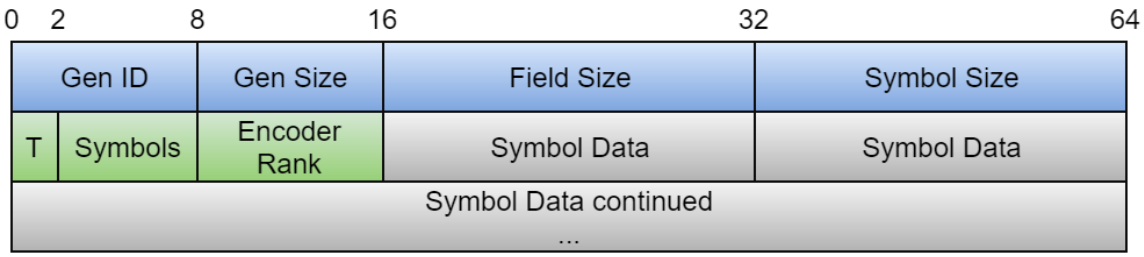


Figure 3.4: A systematic symbol representation.

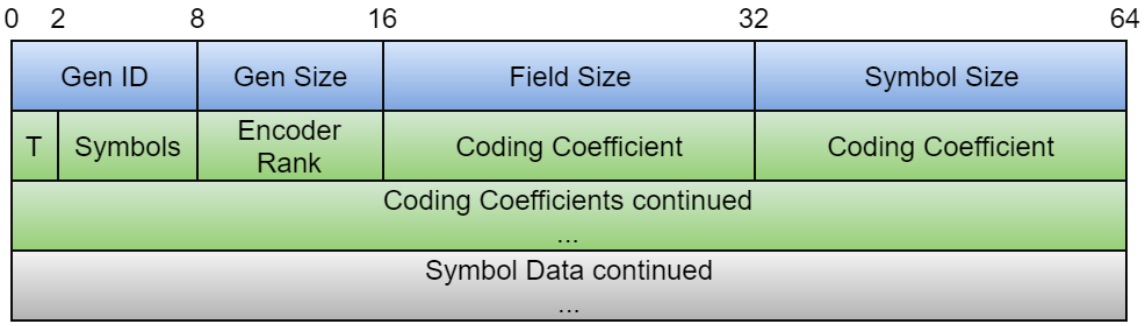


Figure 3.5: A recoded symbol representation.

3.3 Buffering Module

Symbols of the same generation may be carried over multiple packets. Thus, the switch must buffer symbols carried by several incoming packets. When the switch reaches a state where it has collected all the symbols belonging to the same generation, it can start the encoding operation. Yet, the current P4 targets do not have a buffering functionality that easily allows the programmer to buffer the contents of a packet. This limitation spur the need to design a mechanism for buffering packets' symbols and coefficients to be used on future encoding opportunities.

The design of our Buffering Module is based on the following assumptions:

- multiple generations can be simultaneously stored in the buffer.
- all generations have the same size, that is, total number of symbols.
- information pertaining to the same generation, that is, its symbols and coefficients, is always stored in consecutive cells of the same buffer.

The buffering module is made of eight buffers of four different kinds. We need two of each kind to buffer the symbols and the coefficients, respectively. The multiple buffer types are used to ease the simultaneous indexing and storage of information relative to the various generations (recall the 1st assumption above). Due to memory resource constraints, we opted for buffers of finite size. Figure 3.6 illustrates the buffers that constitute the Buffer Module. The first type of buffer, the **Content Buffer**, stores the values of the symbols/coefficients in the packets. All the symbols (or coefficients) that belong to the same generation are stored in consecutive cells (according to the 3rd assumption). The Content Buffer's cells have the same bit-width of the finite field elements, that is, the coefficients and symbols, that it stores.

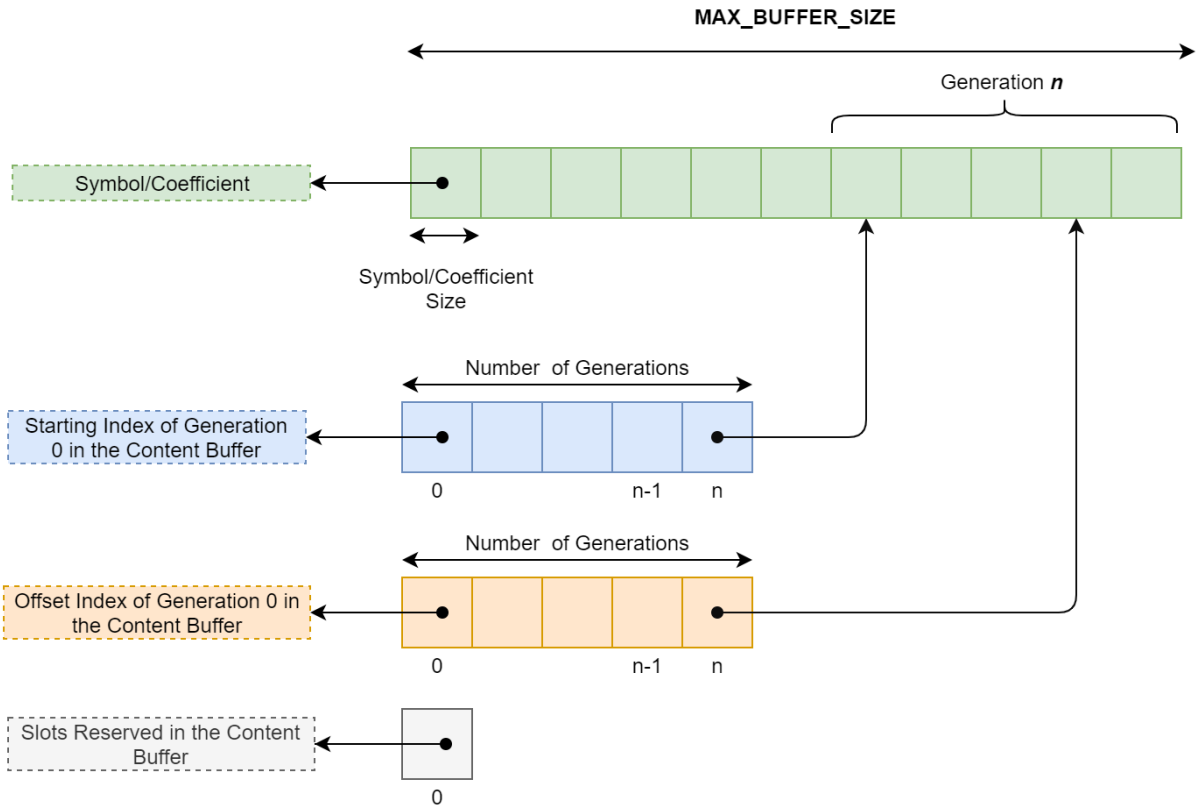


Figure 3.6: The Buffer Module

The second type of buffer, the **Starting Index Buffer**, has a size equal to the number of generations. Every Starting Index Buffer's cell holds the value of the starting index of one generation in the Content Buffer. This buffer is indexed by the generation identification number, one of the fields present in the outer RLNC header. As explained before, this buffer has a major role on discerning when the coding process should start. This will be further expanded in the following sections.

The third type of buffer, the **Offset Buffer**, indicates the number of positions between the starting index of a generation and the next position where a symbol (coefficient) from that generation should be stored.

Finally, the fourth type of buffer, is the **Reserved-Slots Buffer**. This buffer is composed of a single cell that holds the number of slots already used by the currently stored generations for their symbols (coefficients).

As previously mentioned, we use these buffers to dynamically and space-efficiently store multiple generations. In summary, the three buffers we add to the main Content Buffer are used, respectively, to determine where in the Content Buffer a new generation should be stored (Slots Reserved Buffer), and where a generation has been buffered (Starting Index Buffer), where a new element for that generation (symbol or coefficient) should be stored (Offset Buffer).

These buffers essentially form the management part of the Buffer Module, dictating where the packets contents, its symbols and coefficients, should be stored.

The pseudo-code presented in Algorithm 1 describes the procedure to buffer a packet's contents. In lines 2-4, the offset and starting index are fetched from the respective buffers, as well as the numbers of

Algorithm 1 PacketBuffering

```
1:  procedure PACKET_BUFFERING(generation_id, generation_size)
2:      offset_index ← offset_index_buffer[generation_id]
3:      slots_reserved ← slots_reserved_buffer[0]
4:      starting_index ← starting_index_buffer[generation_id]
5:      if offset_index == 0 then
6:          starting_index ← slots_reserved mod length of contents_buffer
7:          if starting_index + generation_size > length of contents_buffer then
8:              drop_packet()
9:          end if
10:         starting_index_buffer[generation_id] ← starting_index
11:         offset_index ← starting_index
12:         slots_reserved_buffer[0] ← slots_reserved + generation_size
13:     end if
14:     buffer_packet(offset_index)
15:     update_offset_index(offset_index)
16: end procedure
```

slots that are already reserved in the Content Buffer. Then, lines 5 to 13 handle the case where symbols from a new generation arrive, and the starting index is calculated. Then, in case the buffer is full, the packet is dropped. Otherwise, the starting index of that generation and the (updated) number of slots reserved are stored to stateful memory. Finally, in lines 14-15, the contents of the packets are buffered and the offset index of the generation is updated.

3.4 Finite Field Operations

Arithmetic to produce linear combinations of the packets' symbols is performed over a Galois Field (GF). A GF is closed over addition and multiplication (Section 2.1.5).

Addition in a $GF(2^n)$ can be performed by logically XOR-ing the operands. Multiplication, on the other hand, is a more complex operation which can be performed through a variety of techniques, offering different performance trade-offs. Since we are dealing with implementing finite field multiplication on a switch data plane, a multiplication algorithm capable of running at line-rate is of the utmost importance.

One way to reduce the cost of multiplication in $GF(2^n)$ is to use precalculated multiplication tables, storing every possible multiplication combination between two elements of the GF. This technique trades additional memory for computational speed. While 2^{16} bytes, or 64KB, are required with $GF(2^8)$ to store the complete multiplication table, 8GB of storage are needed for $GF(2^{16})$. Thus, although improving the speed of multiplication, this techniques introduces a considerable memory overhead. As a switch has only a few tens of MB of stateful memory available [30], this technique cannot be used for large Galois Fields.

As more practical alternatives, we have decided to analyze two different multiplication techniques. First, one algorithm based on Log and Antilog tables, that represents an optimization of the pre-computed tables approach, which reduces the size of the storage required for the tables. Second, a multiplication algorithm that does not require additional memory space, but rather is compute-intensive, by applying shift and add operations iteratively over the operands. We describe these next.

3.4.1 Log/Antilog Tables

Every non-zero element in a finite field can be represented uniquely by a power to a specific primitive element α [31], called the generator of the field. Let $x = \alpha^i, x \in GF(2^n)$, then i is the discrete logarithm of x, with respect to α. Therefore, an antilog/exponential table can be defined to contain each possible power value. For example, x can be represented as:

$$x = \alpha^i = antilog(i) \tag{3.1}$$

Following the same logic, a logarithm table can also be defined as:

$$i = log(x) \tag{3.2}$$

The multiplication between two elements can be represented as follows:

$$x \cdot y = \alpha^i \alpha^j = \alpha^{i+j} \tag{3.3}$$

Which, following the previous definitions, can be rewritten as:

$$a \cdot b = antilog(log(a) + log(b)\ mod\ 2^n - 1) \tag{3.4}$$

Due to the above properties, multiplication in GF can be performed by using two look-up tables, the Log table and the Antilog table, enabling important memory savings. More precisely, while the multiplication tables needs $2^{(n^2)}$ bytes, the Log/Antilog tables simply need 2^n bytes. With $n = 8$, the log/antilog tables require 256 bytes, while for $n = 16$, 2^{16} 2-byte entries are required, or 64kb.

The Equation 3.4 can still be optimized to avoid the modulo operation, which is usually a costly operation and may not be supported across different P4 targets. For that purpose, we have followed the approach used in [32]. The final algorithm used to multiply two elements of a finite field using the lookup tables is shown on Algorithm 2. Lines 2 to 4 handle the case where one of the operands is 0, therefore the returned value is 0 as well. Line 5 sums the log values of both elements. Then, in lines 6-8, if the result value is greater than the defined FLD_SIZE constant minus 1, it is reduced. Finally, in line 9 the result value is used to get the final multiplication result from the antilog table.

Algorithm 2 Multiplying two elements from a finite field using lookup tables without modulo

```
1:  procedure LOOKUP_TABLES_FF_MULT(a,b, FLD_SIZE)
2:      if a == 0 or b == 0 then
3:          return 0
4:      end if
5:      sum ← log[a] + log[b]
6:      if sum ≥ FLD_SIZE − 1 then
7:          sum ← sum − FLD_SIZE − 1
8:      end if
9:      return antilog[sum]
10: end procedure
```

3.4.2 Modified Russian Peasant Multiplication Algorithm

The second approach analysed in our thesis is a Standard Field Multiplication algorithm [33], shown as Algorithm 3. Lines 2 to 7 is the loop to iteratively multiply the elements bit by bit. In line 3, if b is an odd value, then a will be added to the $result$ accumulator. Then, line 4 calculates if a will overflow if we shift it left. In that case, $mask$ will be -1, if not, it will be 0. Line 5, shifts the a variable and reduces it or not depending on the value of $mask$. Then, in line 6, b is shifted right. Finally, line 8 returns the final value of the multiplication which was stored in the $result$ variable.

Conversely to the tables-based algorithm, this technique does not require the allocation of memory space for the multiplication to be performed. This algorithm only performs bit-shifts and XOR operations, operations that are not computationally expensive. However, while it does not require memory accesses, it requires n iterations for a multiplication in $GF(2^n)$.

Algorithm 3 Multiplying two elements using the Modified Russian Peasant Multiplication Algorithm

Input: R is an irreducible polynomial of the finite field and n comes from $GF(2^n)$.

1: **procedure** MODIFIED_RUSSIAN_PEASANT_MULTIPLICATION_ALGORITHM(a,b)
2: **for** i=0 : n - 1 **do**
3: $result \leftarrow result \oplus (-(b \ \& \ 1) \ \& \ a)$
4: $mask \leftarrow -((a >> (n-1)) \ \& \ 1)$
5: $a \leftarrow (a << 1) \oplus (R \ \& \ mask)$
6: $b \leftarrow b >> 1$
7: **end for**
8: **return** result
9: **end procedure**

3.5 Summary

In this chapter we have presented the design of our Network Coding solution, by illustrating its main components and describing their respective functions. The chapter started with an overview of the general architecture and the workflow that a packet undergoes in our switch, in Sec. 3.1. Afterward, a description of the proposed protocol format for Random Linear Network Coding was given in Sec. 3.2. Then, the reasoning for creating a buffering mechanism was presented, along with its design, in Sec. 3.3. Finally, Sec. 3.4 dove into the specifics of finite field arithmetic.

The following chapter will describe the implementation of for our solution, followed by an evaluation of its functionality and performance.

Chapter 4

On the implementation of a Network Coding Switch

In this chapter we present the P4 implementation of our Network Coding switch (we call it RLNC-switch from now on). Our implementation has been driven both by the architectural modules identified in the design-phase and by the physical characteristics of the targeted switch architecture, the Protocol-Independent Switch Architecture (PISA) [4]. This is the reference architecture followed by some P4 targets, including, RMT [34], the P4 reference switch [8], and the Barefoot Tofino [35].

Figure 4.1 illustrates how the architectural blocks of our RLNC-switch have been mapped to a PISA-like switch architecture. The placement of some blocks, namely the buffering and the coding modules, has been dictated by the targeted packet processing behavior. The Buffering Module has been placed in the Ingress Pipeline because buffering is the only processing done to packets when the switch has not yet received a sufficient number of symbols to start coding a certain generation. Until then, every packet's content is buffered in the ingress pipeline and the respective packet is marked to drop by the Forwarding Decision block.

When the buffer for a certain generation is finally full, the switch starts coding that generation. To guarantee that every coded packet will carry innovative information, the coding process has been placed into the egress pipeline where every packet replica generated by the Traffic Manager is processed separately by the egress pipeline. Different random coefficients can be generated for each packet replica in the egress pipeline, to guarantee that every packet contains innovative information, as described above.

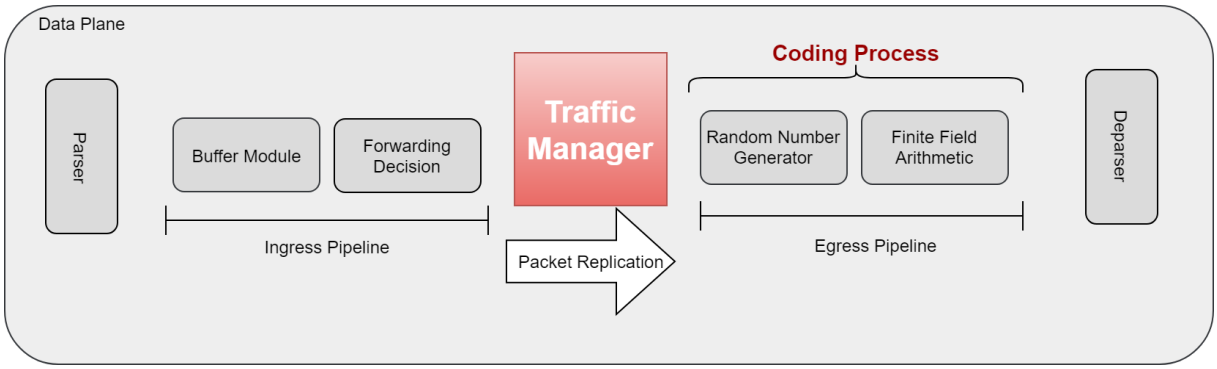


Figure 4.1: RLNC-switch's blocks mapped to a PISA-like switch architecture.

This chapter is organized as follows. The definition of the packet headers and the parser state machine

meant to extract those from incoming packets is presented in Sec. 4.1. A description of the buffering module is then given in Sec. 4.2. Sec. 4.3 explains how the packet replicas are generated for coding. Finally, Sec. 4.4 details how the buffered symbols are coded and forwarded by the RLNC-switch.

4.1 Headers Definition and Parser

The implementation of the headers refers to the packet format proposed in Sec. 3.2. To implement the RLNC headers, we declared four headers in P4, illustrated in **Listing 4.1**:

- An outer header containing information for the buffering purposes, since it pertains to knowledge about the generation of a symbol.
- An inner header that contains information related to the symbols.
- A header for the coefficients, used for recoding purposes.
- A symbol header which is, essentially, the payload of the packet. We treat the payload as a header because it is the only way to manipulate it in a P4 program.

The correct deployment of RLNC in the switch calls for the ability to individually reference the symbols and coefficients of the packet. Therefore, we declared one stack of coefficient headers and one stack of symbol headers.

Listing 4.1: Headers

```
header Rlnc_out_t{
  bit<16> gen_id;
  bit<8>  gen_size;
  bit<16> field_size;
  bit<16> symbol_size;
}

header Rlnc_in_t{
  bit<2> type;
  bit<4> symbols;
  bit<2> padding;
  byte_t encoderRank;
}

header Coeffs_t{
  bit<GF_BYTES> coef;
}

header Symbols_t{
  bit<GF_BYTES> symbol;
}

struct headers{
  Ethernet_t ethernet;
  Rlnc_out_t  rlnc_out;
  Rlnc_in_t rlnc_in;
  Coeffs_t[MAX_COEFFS] coefficients;
  Symbols_t[MAX_SYMBOLS] symbols;
}
```

Figure 4.2 illustrates all the states of our parser. The starting state is the validation/extraction of the ethernet header. If the packet contains an ethernet header with TYPE_RLNC (a previously defined constant) as its etherType, then the switch is receiving traffic for encoding purposes. The following states provide the full validation and extraction of the RLNC outer and inner header.

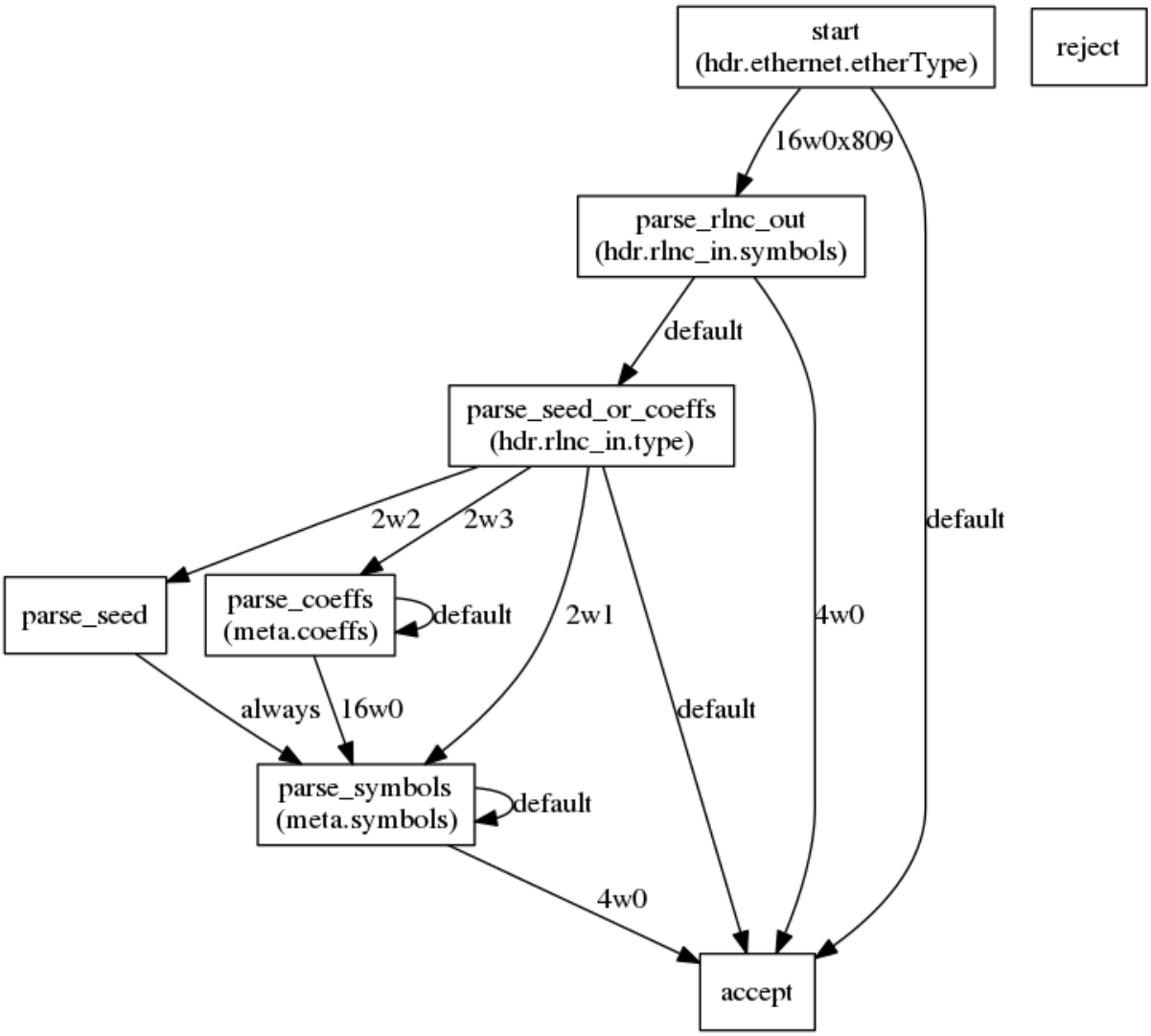


Figure 4.2: Parser of our network coding switch

The validation of the RLNC outer header and inner header consists in the following. The parser extracts the information in the outer header. It then proceeds to validate the inner header. If the symbols field in the inner header equals to 0 the parsing stops. Otherwise, it continues to the next stage where the parser confirms the type of the packet. Type 1 means we are dealing with an uncoded packet where we only need to parse its symbols. If the packet is of type 2 we would need to parse the seed used to generate the coefficients. We recall that currently we do not support this feature. Finally, if the packet is of type 3, we start by parsing the coefficients. Then, we move on to parsing the symbols. Whichever the type of packet, parsing the symbols is always the final stage. Finally, the parsing ends, and the packet moves on to next stage of processing.

Note that in the **parse_seed_or_coeffs** state we save to metadata two values from the header of the packet: the number of symbols and the number of coefficients. Getting the number of symbols from the header is straightforward. Getting the number of coefficients, however, involves some arithmetic operations. According to the definition, the encoder rank expresses the number of symbols over which coding was performed for all coded symbols in the packet. Moreover, it tells us exactly how many coefficients were needed to code a single symbol. Therefore, as each packet can carry one or more

symbols at a time, by multiplying the number of symbols that the packet carries by the encoder rank, we get the total number of coefficients that the packet is carrying.

The two metadata values are used to correctly parse the coefficients and the symbols, which are implemented differently from the rest of the headers. The parser does not have a way to know how many symbols/coefficients are in the packet. To bypass this issue, we created a recursive loop where we parse the symbols/coefficients and decrement the values in the metadata until they reach a value of zero. By doing so, the parsing will not stop until it has finished with all the symbols/coefficients. Only then will the parser move on to the next state.

4.2 Buffering Module

As we mentioned in section 3.3, there is no built-in function to buffer the contents of the packet in the switch. The workaround was to build our own way of buffering them. We accomplished this through the use of registers, a P4 stateful construct that keeps information across packets.

Listing 4.2 shows all the registers used to implement our Buffer Module.

Listing 4.2: Registers

```
// Symbol buffers
register<bit<GF_BYTES>>(MAX_BUF_SIZE)   buf_symbols;
// Coefficient Buffers
register<bit<GF_BYTES>>(MAX_BUF_SIZE)   buf_coeffs;

//Stores and Dictates the index in which a symbol from a specific generation should be stored to
register<bit<32>>(MAX_NUMBER_GENERATIONS) symbol_gen_offset_buffer;
//Stores and Dictates the index in which a coefficient from a specific generation should be stored to
register<bit<32>>(MAX_NUMBER_GENERATIONS) coeff_gen_offset_buffer;

//Buffers the starting index of each generation in the symbol buffer register
register<bit<32>>(MAX_NUMBER_GENERATIONS) symbols_gen_head_buffer;
//Buffers the starting index of each generation in the coefficient buffer register
register<bit<32>>(MAX_NUMBER_GENERATIONS) coeff_gen_head_buffer;

//Keeps track of the number of symbol slots reserved by all generations
register<bit<32>>(1) symbol_slots_reserved_buffer;
//Keeps track of the number of coefficients slots reserved by all generations
register<bit<32>>(1) coeff_slots_reserved_buffer;
```

We used eight registers to implement this module. As seen in section 3.3, there are four types of buffers. By going through the registers two by two, the reader can see that we have registers of type Contents Buffer, Offset Buffer, Starting Index Buffer and Slots Reserved Buffer, respectively.

Listing 4.3 is Algorithm 1, written in the P4 language. We implemented the entire buffering mechanism on the ingress pipeline. More precisely, we placed it in the control block. The listing below shows how we make use of each register to coordinate the buffering of the symbols of a packet.

Listing 4.3: Control flow of symbol buffering

```
if((hdr.rlnc_in.type == 1 || hdr.rlnc_in.type == 3)) {

    // loading the buffer index for the current generation
    symbol_gen_offset_buffer.read(symbols_gen_offset, (bit<32>)gen_id);

    // loading the number of slots that were already reserved by all generations so far (buffer shared
     across diff generations)
    symbol_slots_reserved_buffer.read(symbols_reserved_slots, 0);
    // loading the starting index of the generation
    symbols_gen_head_buffer.read(symbols_gen_head, (bit<32>) gen_id);

    // the storing of new generation is based on the free buffer space
    if(symbols_gen_offset == 0) {
```

```
        symbols_gen_head = symbols_reserved_slots % MAX_BUF_SIZE;

        if(symbols_gen_head + gen_size > MAX_BUF_SIZE) {
            my_drop();
            return;
        }
        symbols_gen_head_buffer.write((bit<32>)gen_id, symbols_gen_head);
        symbols_gen_offset = symbols_gen_head;

        // incrementing the number of slots reserved
        symbol_slots_reserved_buffer.write(0, symbols_reserved_slots + gen_size);
    }

    // Using the generation index to save to the registers the packet symbols
    action_buffer_symbols();

    // incrementing the symbols_gen_offset
    action_update_symbols_gen_offset();
```

We start by loading three values, the offset and starting index for the current generation, and the number of slots all generations already reserved so far. The next step is to know if this is the first time that the switch is seeing this generation or not. This is done by verifying if the offset index has already been attributed a value.

If not, we are dealing with an unseen generation. In which case, we must know where to start indexing the generation in the Contents Buffer. To decide the value of the starting index, we resort to the slots reserved value. This gives the index to the next free position in the buffer. The modulo operation is there to ensure that the indexation loops back to the beginning of the buffer when it reaches full capacity. We then reach another branching phase. If the buffer does not have enough space to store an entire generation, then we drop the packet. Otherwise, we continue on with the buffering process. Now that we have computed the starting index, we need to save that value to stateful memory so it can be used again over multiple packets. We also increment and save the slots reserved value according to the size of the generation.

Either if it is the first time seeing a generation or not, the last step is always to buffer the symbols of the packet using the current offset index and increment it. **Listing 4.4** shows the two auxiliary actions that enable us to do so. The **action_buffer_symbols** stores the symbols to the registers by accessing the declared stack of the symbols header field and writing it into the corresponding register using the current offset index. Then, the **action_update_symbols_gen_offset** increments the offset index based on the number of symbols that the packet was carrying (in the example shown, only 1 symbol). Finally, we save the value of the offset index to stateful memory so we can use it over multiple packets belonging to the same generation. There is no corresponding listing showing the code to buffer the coefficients due to its similarity. The only notable difference is that the switch will only buffer the coefficients of a packet when its header field `hdr.rlnc_in.type` has a value of 3.

Listing 4.4: Writing the symbols to the register and incrementing the index

```
action action_buffer_symbols() {
    buf_symbols.write(symbols_gen_offset + 0, hdr.symbols[0].symbol);
}

action action_update_symbols_gen_offset() {
    symbols_gen_offset = symbols_gen_offset + numb_of_symbols;
    symbol_gen_offset_buffer.write((bit<32>)gen_id, symbols_gen_offset);
}
```

4.3 Packet Replication

We perform packet replication by leveraging a packet replication primitive, the multicast mechanism. Traditionally, when using this primitive, we start by creating a multicast group. Then, we create the nodes paired with various ports and associate those same nodes with the multicast group. Finally, we set the `standard_metadata.mcast_grp` (**Listing 4.6**) field of the packet to the multicast group created. The Traffic Manager will then replicate that packet to be sent to all the output ports that were associated with the nodes.

As **Listing 4.5** shows, we associate all nodes with the same port. In doing so, the Traffic Manager will replicate the packet, but every replica will be sent through the same output port. By leveraging this primitive we achieve the capacity of generating multiple packets with different linear combinations.

Listing 4.5: Multicast group creation

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

Listing 4.6: Setting the multicast group metadata

```
action action_clone(bit<16> group) {
    standard_metadata.mcast_grp = group;
}
```

4.4 Coding Process

After the packet replication occurs, the packet enters the egress pipeline, where the coding process starts. The Random Number Generator is the first module the packet goes through. The switch starts by generating the random coefficients using a random function and then stores them to global variables (**Listing 4.7**). They, in turn, are used to code the symbols, which are loaded from the registers (**Listing 4.8**).

Recall that the coding process may branch, depending on the value of the type header field. If the switch is dealing with an uncoded packet, it still needs to append the random coefficients (**Listing 4.9**) to the header and update the value of the encoder rank header field. In Standard RLNC, this header field value is always equal to the generation size. This is because we perform coding over all the symbols of a generation. Finally, the type header field is modified to 3, meaning that the packet is now a coded/recoded packet.

Listing 4.7: Generating the random coefficients

```
action action_generate_random_coefficients() {
    bit<GF_BYTES> low = 0;
    bit<GF_BYTES> high = GF_MAX_VALUE;
    random(rand_num0, low, high);
    random(rand_num1, low, high);
    random(rand_num2, low, high);
    random(rand_num3, low, high);
}
```

Listing 4.8: Loading the symbols

```
action action_load_symbols(bit<32> idx) {
    buf_symbols.read(s0, idx);
    buf_symbols.read(s1, idx + 1);
    buf_symbols.read(s2, idx + 2);
    buf_symbols.read(s3, idx + 3);
}
```

Listing 4.9: Appending the random coefficients to the headers

```
action action_add_coeff_header() {
    hdr.coefficients.push_front(4);
    hdr.coefficients[0].setValid();
    hdr.coefficients[1].setValid();
    hdr.coefficients[2].setValid();
    hdr.coefficients[3].setValid();
    hdr.coefficients[0].coef = rand_num0;
    hdr.coefficients[1].coef = rand_num1;
    hdr.coefficients[2].coef = rand_num2;
    hdr.coefficients[3].coef = rand_num3;
    hdr.rlnc_in.encoderRank = (bit<8>) gen_size;
}
```

The switch can receive a packet containing already coded symbols. This is verified by checking if the type header field has a value of 3. If it is the case, then the switch applies recoding. Recoding implies not only coding the symbols but also updating its coefficients using the same random generated coefficients used to code the symbols. This is the basic coding process that the packet goes through. **Listing 4.10** illustrates all the steps required to fully code the symbols and the coefficients of a packet.

We also implemented a mechanism, at the very end of the egress pipeline, that allows us to know when the established number of generated coded packets have been processed. We do this by using a register to count how many packets have gone through the egress pipeline. When the value saved in the register is greater or equal than **meta.clone_metadata.n_packets_out**, we free the space reserved by the current generation (**Listing 4.11**). This metadata holds the value of how many coded packets should be generated, regarding a single generation, before the switch can free the space.

Listing 4.10: Egress Pipeline

```
if(hdr.rlnc_in.isValid() && meta.clone_metadata.coding_flag ==1 && meta.rlnc_enable == 1) {
    action_generate_random_coefficients();
    #Placeholder: This is where we perform the arithmetic to code the symbols
    if(hdr.rlnc_in.type == 1) {
        // adding the coefficient vector to the header
        action_add_coeff_header();
        // Since we coded the packet we change its type to TYPE_CODED_OR_RECODED
        action_systematic_to_coded();
        }
    else if(hdr.rlnc_in.type == 3) {
        #Placeholder: This is where we perform the arithmetic to code the coefficients
    }
    packets_sent_buffer.read(packets_sent, 0);
    packets_sent = packets_sent + 1;
    if(packets_sent >= meta.clone_metadata.n_packets_out) {
        action_free_buffer();
        packets_sent_buffer.write(0,0);
    }else {
        packets_sent_buffer.write(0, packets_sent);
    }
}
```

Listing 4.11: Freeing register space

```
action action_free_buffer() {
    bit<32> tmp = meta.clone_metadata.symbols_gen_head;
    buf_symbols.write(tmp, 0);
    symbol_gen_offset_buffer.write((bit<32>)hdr.rlnc_out.gen_id, 0);
    coeff_gen_offset_buffer.write((bit<32>)hdr.rlnc_out.gen_id, 0);
    symbols_gen_head_buffer.write((bit<32>)hdr.rlnc_out.gen_id, 0);
    coeff_gen_head_buffer.write((bit<32>)hdr.rlnc_out.gen_id, 0);
}
```

Placeholders markers are present in the listing, instead of the actual code, because the arithmetic operations can differ depending on the approach employed. We will expand upon these implementation approaches in the following sections. Our design entails two different implementations regarding the finite field operations. Therefore, we implemented two slightly different egress pipelines, one for the lookup tables

(Section 3.4.1) and another for the Modified Russian Multiplication Algorithm (Section 3.4.2). In the following sections, we will describe each one of them and show the code that is missing in **Listing 4.10**. The examples shown will only relate to the symbols and not the coefficients because the implementations are very similar. Therefore, we will only present the code from the symbol side.

4.4.1 Lookup Tables Egress Pipeline

As we mentioned in Section 3.4.1, the Log/Antilog tables have to be pre-computed and saved to stateful memory in the switch during the start-up phase. We do this through the `register_write register_name index value` CLI command. We save both tables into two registers, one for each table (**Listing 4.12**). The constants **GF_BYTES** and **GF_BITS** come directly from the Galois Field chosen. The former determines the bit-width size of the elements stored in it, while the latter determines the number of cells in the register. For example, if we are dealing with $GF(2^8)$, the **GF_BYTES** would be equal to 8 and **GF_BITS** to 255.

Listing 4.12: Log/Antilog registers

```
register<bit<GF_BYTES>>(GF_BITS)          log_table;
register<bit<GF_BYTES>>(GF_BITS)       antilog_table;
```

In the case of the lookup tables egress pipeline, the bulk of the finite field multiplication is in the control block. The reason is that our switch target, the software switch Bmv2, does not support certain conditional executions inside an action. More precisely, it does not allow a conditional execution involving reads from a register. Therefore, our solution was to place all the if conditions in the control block and the reads inside actions. **Listing 4.13** shows a code snippet of the multiplication process between a symbol and a random coefficient. Only a portion of it is shown but, to give an example if we were dealing with generations of size 4 we would have this piece of code replicated 4 times, for every corresponding symbol and random coefficient.

Listing 4.13: Multiplication of two elements

```
if(s0 != 0 && rand_num0 != 0) {
    action_get_sum_log(s0, rand_num0);
    if(sum_log >= 255) {
        sum_log = sum_log - (255);
    }
    action_get_antilog_value_and_add(sum_log);
}
.
.
.
hdr.symbols[0].symbol = lin_comb;
```

Multiplying a pair of elements is as follows. First, we only perform the multiplication if and only if the two elements differ from zero. When that condition is met, we get the log values of both elements from the log register, and sum them together (**Listing 4.14**); If the resulting value is larger than the maximum value of the finite field minus one (in the example shown we consider $GF(2^8)$), we reduce it. Finally, we use the result obtained to get the multiplication result from the antilog register. We then add it to the variable lin_comb (**Listing 4.15**) which, after every multiplication between symbol and random coefficient, will hold the value of the coded symbol. Notice how finite field addition is performed using the XOR operation. The last step is to attribute to the corresponding symbol header the value in the variable lin_comb. This closes the process of generating a coded symbol.

Listing 4.14: Getting the sum of the log values

```
action action_get_sum_log(bit<GF_BYTES> x, bit<
    GF_BYTES> y) {
    bit<GF_BYTES> tmp_log_a = 0;
    bit<GF_BYTES> tmp_log_b = 0;
    bit<32> log_a = 0;
    bit<32> log_b = 0;

    GF256_log.read(tmp_log_a, (bit<32>) x);
    GF256_log.read(tmp_log_b, (bit<32>) y);

    log_a = (bit<32>) tmp_log_a;
    log_b = (bit<32>) tmp_log_b;
    sum_log = (log_a + log_b);
}
```

Listing 4.15: Getting the antilog value and adding it to the final result

```
action action_get_antilog_value_and_add(bit<32>
    log_sum) {
    bit<GF_BYTES> mult_result = 0;
    GF256_invlog.read(mult_result, log_sum);
    lin_comb = lin_comb ^ mult_result;
}
```

4.4.2 Modified Russian Algorithm Egress Pipeline

In this approach we did most of the work through actions. Therefore, leaving the control block in a more "clean" state than in the previous approach. The action shown in **Listing 4.16** is the starting point, called in the control block, to the process of coding a symbol. This one, in turn, calls **action_GF_arithmetic**, which triggers the arithmetic operations.

Listing 4.16: Finite Field Arithmetic

```
action action_code_symbol() {
    action_GF_arithmetic(s0, rand_num0, s1, rand_num1, s2, rand_num2, s3, rand_num3);
    hdr.symbols[0].symbol = lin_comb;
}
```

The **action_GF_arithmetic (Listing 4.17)** illustrates the key difference to the lookup tables approach. Remember that the Modified Russian Peasant Multiplication is an iterative algorithm. This presented a challenge in its implementation, since P4 by nature does not easily allow the construction of loops. The solution was to call the same action n times, where n comes from $GF(2^n)$, and carry the values over each iteration.

Listing 4.17: Finite Field Arithmetic

```
action action_GF_arithmetic(bit<GF_BYTES> x0, bit<GF_BYTES> y0, bit<GF_BYTES> x1, bit<GF_BYTES> y1, bit<
    GF_BYTES> x2, bit<GF_BYTES> y2, bit<GF_BYTES> x3, bit<GF_BYTES> y3) {
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_ffmult_iteration();
    action_GF_add(mult_result_0, mult_result_1, mult_result_2, mult_result_3);
}
```

Listing 4.18 shows the implementation of a single iteration of the Algorithm 3. However, one issue arose regarding the operation with the IRRED_POLY constant, which is of type int. According to the P4-16 specification [24], applying a binary operation to an expression of type int and an expression with a fixed-width type will implicitly cast the int expression to the type of the other expression. Considering a $GF(2^8)$, the constant has a value of 283, which does not fit in a bit<8> type. Therefore, in this case, there is an implicit cast of IRRED_POLY to bit<8> before performing the AND operation. The cast takes only one bit of the constant, which gives us a value of 27. Thus, we changed the polynomial value to a bit<8> value. This way we assure that by moving the program to a different target (and compiler toolchain), we will not get any unexpected result.

Listing 4.18: Multiplication of two elements

```
bit<GF_BYTES> mask = 0;
mult_result_0 = mult_result_0 ^( -(tmp_y0 & 1) & tmp_x0);
mask = -((tmp_x0 >> 7) & 1);
tmp_x0 = (tmp_x0 << 1) ^ (IRRED_POLY & mask);
tmp_y0 = tmp_y0 >> 1;
```

4.5 Summary

This chapter presented the details of the implementation of our solution, describing its various components and their role. In particular, it described the headers and parser in Sec. 4.1, followed by a detailed walkthrough of the packet processing implementation. The implementation of packet buffering was described in Sec. 4.2, while achieving packet replication was presented in Sec. 4.3. Finally, Sec. 4.4 describes the implementation of the coding process while its subsections, Sec. 4.4.1 and Sec. 4.4.2, describe the implementation of the finite field multiplication algorithms.

The next chapter presents a preliminary evaluation of the correctness of the coding operations and of the network performance of our RLNC-switch.

Chapter 5

Evaluation

This chapter describes the evaluation performed to validate the functionality and performance of our RLNC-Switch. In a Generation-Based RLNC solution, the coding parameters have a very critical impact on the device employing it. They can affect the throughput, the CPU power consumption, and packet losses. Therefore, we devoted our time to studying this aspect. The evaluation consists of two main parts:

- An analysis of how changing the coding parameters affects the program developed in terms of lines of code and size of the compiled file.
- A study of the performance of our solution with different coding parameters.

The rest of this section is organized as follows. Sec. 5.1 gives a description of the test environment and an explanation of the tools developed for the evaluation. Sec. 5.2 presents the evaluation on the proposed solution. Within this section, Subsec. 5.2.1 presents an analysis of the developed program, Subsec. 5.2.2 presents the validation of the solution, and Subsec. 5.2.3 presents the performance evaluation.

5.1 Development & Testing Environment

All the experimental tests were run on a machine from the Quinta at the Faculty of Sciences of the University of Lisbon. Quinta [36] is a processor farm. It is a computational cluster dedicated to large-scale experiments of distributed systems. It is currently comprised of 42 physical machines, which compose a test bed with more than 300 processing cores, 1.3 TB of RAM and 33 TB of storage. More specifically, our experiments were run on a bare-metal Dell PowerEdge R440 server with 2x Intel Xeon Silver 4114, 2.2GHz, and 32GB of memory that is part of Quinta. For the purpose of this evaluation, some specific tools have been developed within this project, which are presented through the rest of this section.

5.1.1 P4 Code Generator

One of the main goals of this work was to see the importance of the coding parameters in the RLNC-Switch. More precisely, to study how our implementation would perform with different configurations. However, it is not easy for any P4 program to adapt itself to different inputs without some "outside" help. For that reason, we developed a Python script that takes the coding parameters and a P4 program template as input, and outputs a P4 program accordingly. Other than the coding parameters, the Python script also receives additional input to facilitate setting up the experimental tests. The parameters are the following:

- `-t <type>`, sets the type of the packet, can only be 1 or 3. Default is 1 (see Sec. 3.2).
- `-f <field_size>`, sets the size of the finite field.
- `-g <gen_size>`, sets the generation size to gen_size.
- `-p <n_packets>`, sets the number of packets to be sent to n_packets.
- `-s <n_symbols>`, sets the number of symbols per packet to n_symbols.
- `-m <mul_type>`, sets the type of multiplication to mul_type: 1 is multiplication through log/antilog tables, 2 is multiplication through the Russian Peasant Multiplication Algorithm.
- `-c <lin_comb>`, sets the number of packets generated by the switch to lin_comb.
- `-pps <pps>`, sets the sending rate of packets for the sender application.

Additionally, this python scripts outputs a config.txt file for the end-host applications to use as input so the coding parameters can be agreed upon between both sender and receiver application.

5.1.2 End-Host Applications

Scapy [37] is a packet manipulation tool for computer networks. It allows the user to send, capture and forge packets. It can also enable various network tasks such as scanning, tracerouting, probing, unit tests, attacks, and network discovery. The most relevant aspect of Scapy for this work was the ability to forge packets. Since RLNC is not a standard internet protocol, there is no packet header format defined for it. Scapy enabled the design of a custom packet header format for RLNC. Using these custom-built packets it was possible to produce traffic to test and evaluate our RLNC-Switch.
We have developed three python applications: one sender and two receivers. They were designed with the purpose of evaluating the functionality and performance of our implementation. All three receive the config.txt file as input to set up the coding parameters between them.

The sender application constructs a list of packets based on the type of packet, the generation size, the number of symbols per packet, the field size, and the total number of packets to be sent. It groups the packets within generations and sends them over the network at the rate defined. This application was designed to function as a traffic generator to stress test the RLNC-Switch.

The decoder application was developed to test the correctness of the P4 program. It receives the packets from the sender and performs the decoding process to obtain the original symbols with the help of the python package found in [38].

The measure application has the purpose of collecting data regarding the receiving rate, the CPU usage and how many packets were lost. It then saves that data to stateful memory to be used to plot the results.

5.2 Evaluation of the Network Coding Switch

In this section, we first present a preliminary evaluation of our P4 program. Then we demonstrate its correct functionality. We then follow with a study of the performance of the different multiplication

approaches. Finally, we evaluate the impact that the coding parameters have on our network coding switch, the cost that recoding introduces, and we pinpoint where packet losses occur.

5.2.1 Program Analysis

The coding parameters, including the generation size and the number of symbols in the packets, have an impact on the lines of code of the program and on the resulting compiled file. Most notably, it has an impact on the buffering and coding modules. Therefore, we used different configurations of these parameters and measured the size of the corresponding P4 programs. We tested both multiplication algorithms as their implementations were slightly different. From now on, x in Gx represents the size of the generation while y in Sy represents the number of symbols in the packet.

	Alg1 - Lookup Tables		Alg2 - MRPA	
	LoC	File Size (Mb)	LoC	File Size (Gb)
G8S4	2761	4.2	748	0.9
G16S4	8985	15	1166	4
G16S8	17061	30	1640	7.9
G32S4	32963	57	1852	19

Table 5.1: Comparison of the lines of code and compiled file size of the different multiplication approaches.

From Table 5.1, we observe that the Lookup Tables approach is the most affected in terms of lines of code (LoC) by the increase in the coding parameters. This result comes from the fact that most of the finite field multiplication had to be implemented directly in the control block. The fact that the software switch does not support registers reads within actions with conditional executions did not allow us to reuse code. In contrast, the Modified Russian Peasant Algorithm (MRPA) approach does not require reads nor conditional executions. This characteristic allowed us to reuse code, by calling an action with different parameters, to multiply a pair of elements.

The most interesting result, however, was how increasing the coding parameters affected the size of the compiled file. Even though the Modified Russian Peasant Algorithm generates a much inferior number of LoC, its generated file size is 3 orders of magnitude larger. We are still investigating the root cause of this difference. We are trying to understand if a different programming style would reduce it, and are also looking deeper at the target compiler. One possibility is the fact that this algorithm performs bit-by-bit operations that may be a poor fit for the target architecture.

5.2.2 Functionality Test

We performed a set of basic experiments to validate our solution. This entails showing that a switch installed with our P4 program can both code and recode an entire generation, while the receiver can successfully decode the generation and obtain the original symbols sent. What follows is an example where we follow the process of creating a generation (consisting of systematic symbols) at a source node (h1); coding it at the first intermediate node (s1); recoding it at the second intermediate node (s2); and finally decoding it at the destination node (h2). Figure 5.1 illustrates the topoly used in the functionality test.

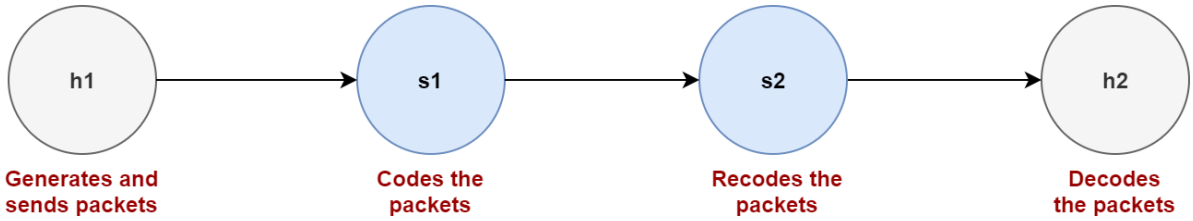


Figure 5.1: Topology used in the functionality test.

The configuration parameters are: a $GF(2^8)$, a generation size of 8 symbols, and 8 packets sent with 1 symbol each.

Figure 5.2 illustrates the full process of performing RLNC over a generation. To help the visualization, all the matrices presented in the figure, were built with the information extracted from the packets. In this case, their symbols and coefficients.

The source host in Figure 5.2a starts by generating symbols at random and sends eight packets with one symbol each. When the first switch has received all eight symbols it initiates the coding process. The coding switch generates a total of eight packets with one coded symbol, each with its coefficient vector (a row of the random coefficient matrix seen in Figure 5.2b). The recoding switch follows the same behavior, it waits until it has collected a full generation and then it starts the recoding process. Recoding generates recoded symbols, but it also updates the coding vectors. As illustrated by Figure 5.2c, both the random coefficient matrix and the symbols are different from those in Figure 5.2b. Finally, when all eight packets arrive at the destination host, it can decode and obtain the original symbols. The symbols obtained in Figure 5.2d are the same as the originally sent in Figure 5.2a. With this, we demonstrate the correct functionality of our network coding switch.

5.2.3 Performance Tests

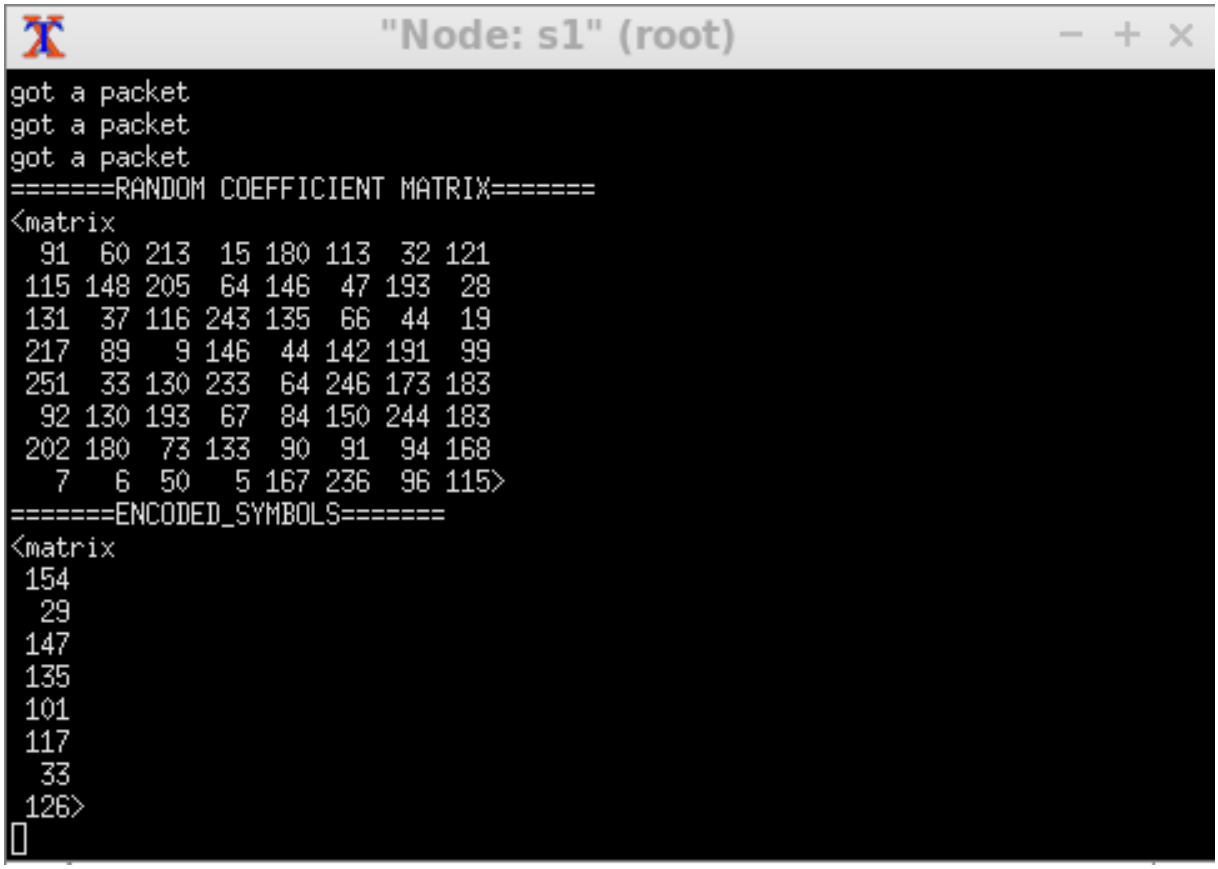
We set up an experiment where we evaluate our network coding switch performance while varying the coding parameters. For all the reported results, ten independent experiments have been executed and the average values are plotted. Through this evaluation, we have tried to assess i) differences in the performance between the two multiplication algorithms we implemented, ii) the impact of coding parameters on the switch performance, iii) the performance penalty of performing re-coding and iv) pinpoint exactly where, in the switch pipeline, are packets being lost.

The performance of the RLNC-Switch was evaluated with the topology presented in Figure 5.3. Each experiment consists of the following. The source node, *h1*, injects traffic at a specific rate. In the destination node, we collect the metrics and compute their average and standard deviation.

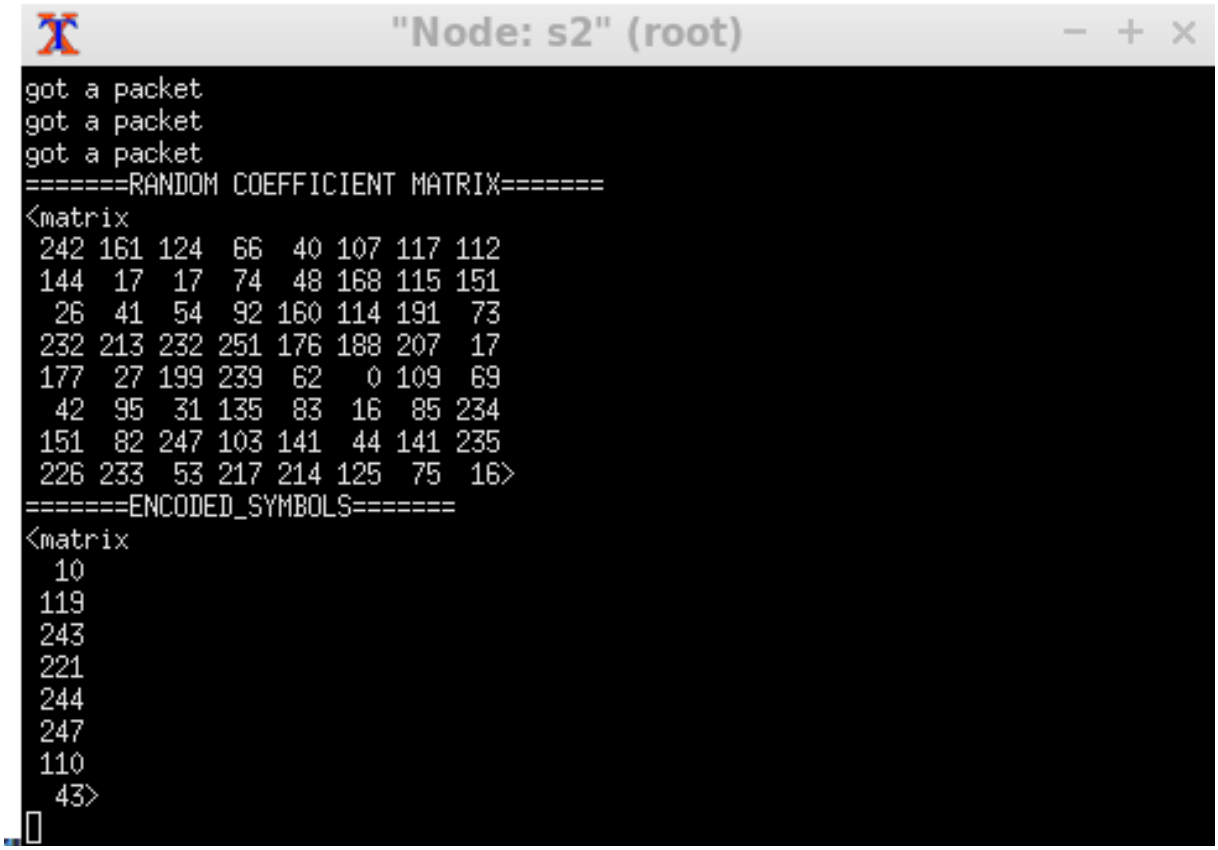
The metrics collected were: **Throughput**, **CPU Utilization** and **Packets Loss**. To measure the throughput, we count the number of received packets, per second, on the interface of the destination node. We measure the CPU utilization to see how compute intensive is our solution. We count the number of packets that arrive at the destination node and subtract it to the value that should have arrived in total to measure the number of packets lost.

The first step in the evaluation was to determine which of the multiplication approaches was the most promising and therefore, worth pursuing with further experiments. The experiments plotted, as seen in **Figure 5.4**, illustrate the difference in performance of both algorithms. We see that in every metric

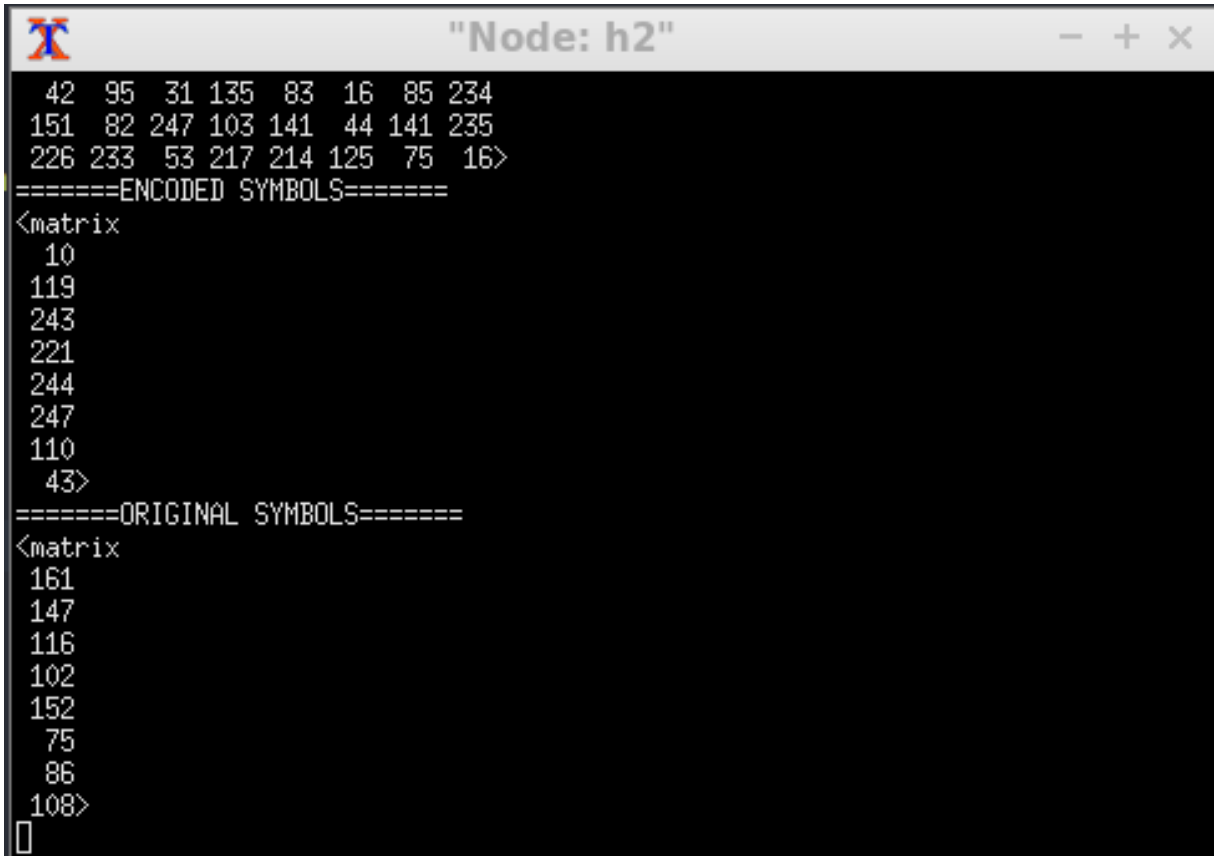
(a) Source Host



(b) Coding Switch



(c) Recoding Switch



(d) Destination Host

Figure 5.2: A full example of RLNC

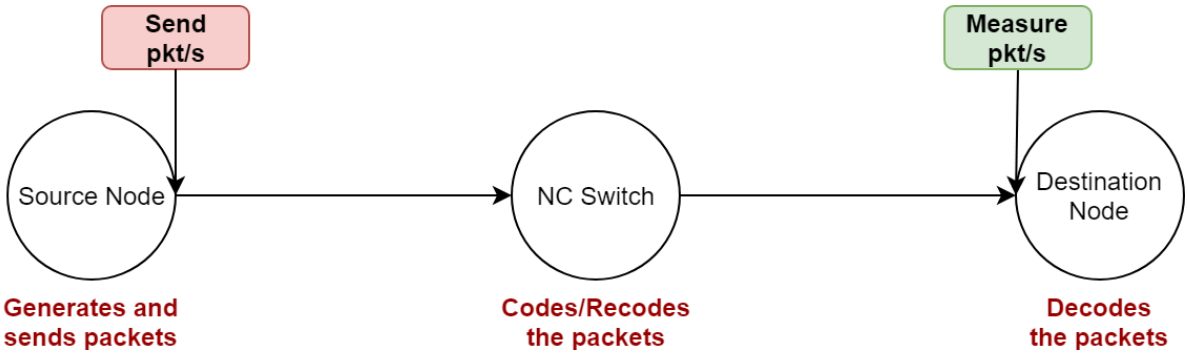


Figure 5.3: Topology used for the evaluation.

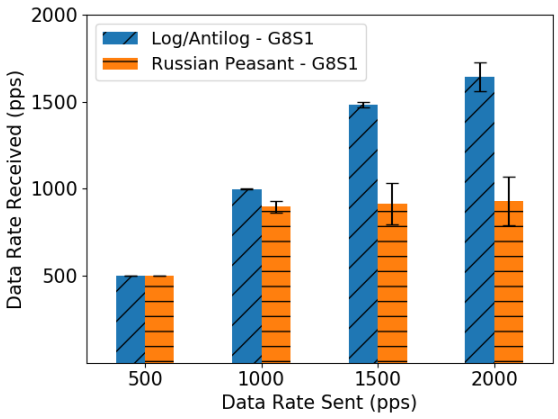
plotted, be it in Figures **5.4a**, **5.4b** or **5.4c**, Algorithm 2 (Log/Antilog Tables) outperforms Algorithm 3 (Modified Russian Algorithm). The former can handle higher values of data rate, it loses fewer packets, and consumes less CPU. The latter is more compute-intensive as expected. It is, by nature, an iterative algorithm and it uses operations such as additions, shifts, logical-and, which prove to be much more costly than register reads. Doing the same experiments while increasing the coding parameters, or even performing recoding, would result in even lower performance from Algorithm 3, due to the increase in the computational complexity which comes from an increased number of multiplications. These results, together with those from Table 5.1, regarding the compiled file size, led us to only further evaluate the effect of the coding parameters on **Algorithm 2**, the Log/Antilog tables approach.

An increase in the coding parameters should, theoretically, negatively impact the performance of the network coding switch due to, for example, the increase in computation complexity that larger generation sizes introduce. **Figure 5.5** confirms our assumptions. An increase x of the generation size determines an increase of the coding vector per each symbol and, by consequence, an increase of a factor of $x \cdot n$ on the number of required arithmetic operations, where n is the number of symbols per packet. To give a clear picture, we enumerate the exact number of multiplication for the different configurations:

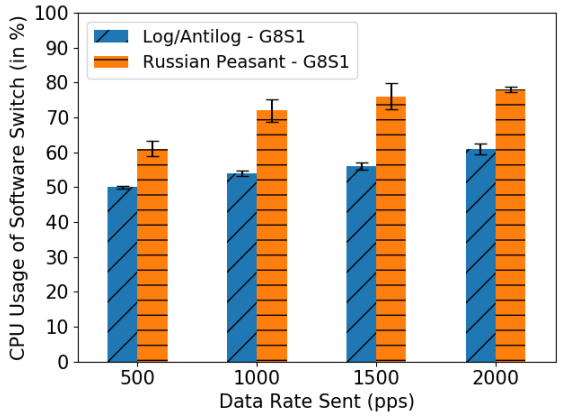
- **G8S4:** $8 \cdot 4 = 32$ multiplications
- **G16S4:** $16 \cdot 4 = 64$ multiplications
- **G16S8:** $16 \cdot 8 = 128$ multiplications
- **G32S4:** $32 \cdot 4 = 128$ multiplications

An interesting observation is that configurations ***G16S8*** and ***G32S4*** regardless of performing the same number of multiplications, have different performance: ***G32S4*** performs worse. This happens because a larger generation size does not only increase the number of total multiplications to perform, but also increases the number of symbols that need to be loaded from the buffer, and the number of randomly generated coefficients, and consequently the number of coefficients that need to be appended to the packet header.

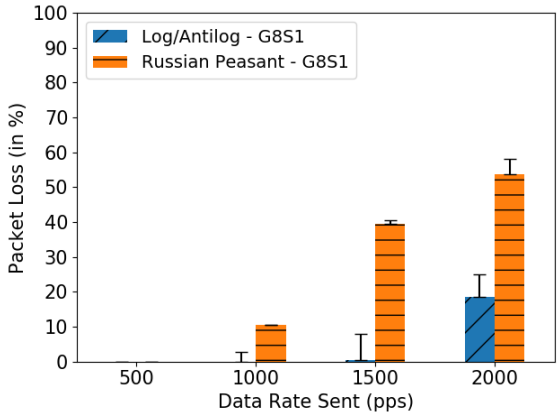
In conclusion, the total number of multiplications is not the only factor influencing the network coding switch performance, but actions such as loading symbols from the registers, generating random coefficients and appending them to the packet header, also have an important impact.



(a) Throughput in packets per second



(b) CPU usage (in %).



(c) Packet Loss (in %)

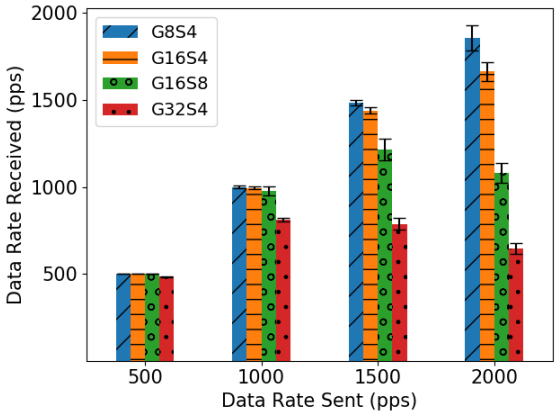
Figure 5.4: Results from employing coding with the two multiplication approaches with generation size 8 and 1 symbol per packet

Next, we look at recoding, to observe it to be a more costly operation than coding. **Figure 5.6** illustrates the difference in performance between the two, while using the same configuration: a generation size of 8 and 4 symbols per packet. Other than the fact that recoding requires additional arithmetic over the coding vectors, the switch also has to parse and buffer, in addition to the symbols in the packet, the coefficients. We break down the number of multiplications and how many elements are parsed and buffered for coding and recoding:

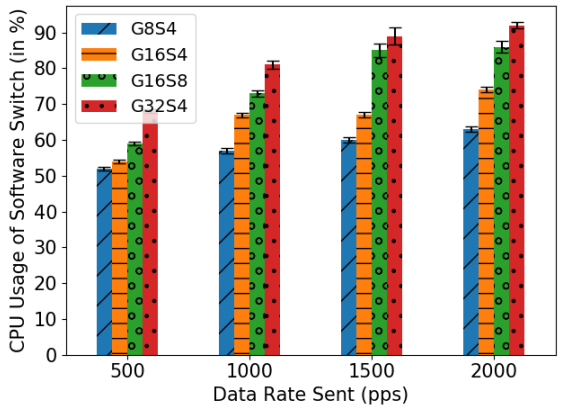
- **G8S4-Coding:** 32 multiplications; 4 elements parsed and buffered per packet;
- **G8S4-Recoding:** 288 multiplications; 36 elements parsed and buffered per packet;

Recoding, in this case, performs 9 times more multiplications and parses/buffers 9 times more elements. This justifies the difference in the performance results.

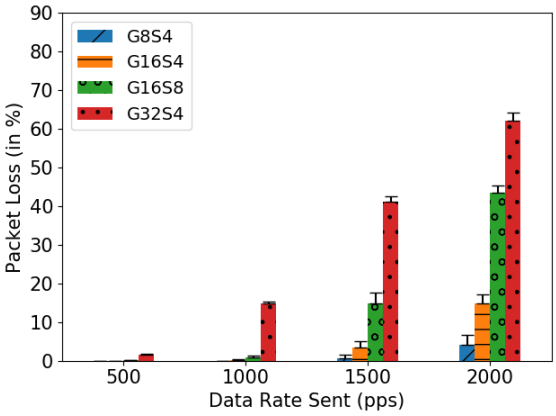
By analyzing the plots so far we can conclude that packet loss is a problem that persists in higher data rates and configurations with higher values. A more intriguing question arose from this problem. Where exactly are packets being lost? To find the answer to that question we performed an experiment where we stress tested the switch with a data rate of **2000** packets per second. **Figure 5.7** maps to each section of the switch pipeline the packet loss percentage. We can see that for the simpler configurations packets are



(a) Throughput in packets per second

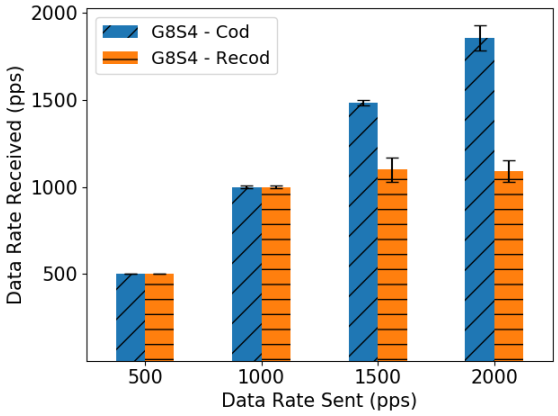


(b) CPU usage (in %).

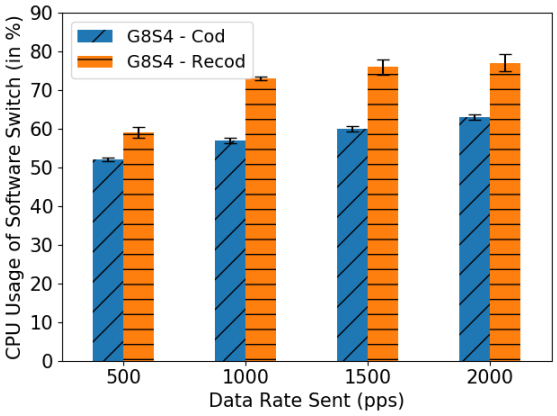


(c) Packet Loss (in %)

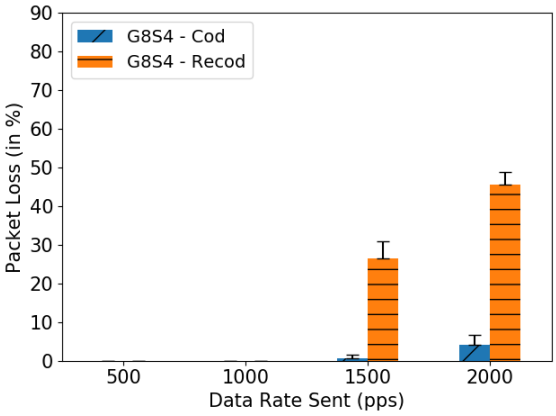
Figure 5.5: Results from exploring different coding configurations



(a) Throughput in packets per second



(b) CPU usage (in %).



(c) Packet Loss (in %)

Figure 5.6: Differences in performance between coding and recoding operations

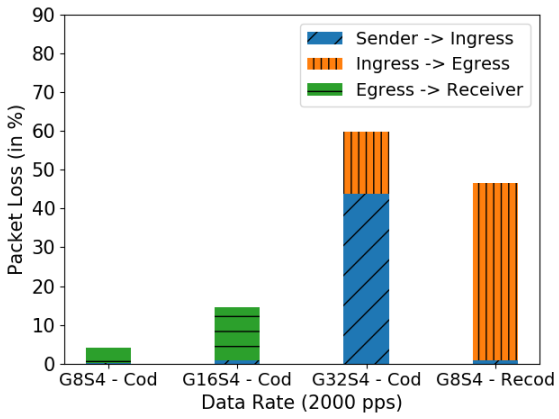


Figure 5.7: Pin-pointing the packet loss in the switch pipeline

lost somewhere between exiting the egress pipeline and arriving at the receiver. For the most complex configuration (G32S4-Cod), most of the loss happens between the sender and the ingress. In the case of recoding (G8S4-Recod), even with low coding parameters, there is a loss of almost half of the total packets sent, happening mostly between the ingress and egress pipeline. In both cases, with G32S4-Cod and G8S4-Recod, there is no loss at *Egress -> Receiver*. This is due to this section not being under so much workload, since the bottleneck in this case occurs in prior modules.

We break down what is, most likely, causing packet losses in each of the sections:

- ***Egress -> Receiver*:** happens due to working on a software switch that is bound to the CPU: the switch can not emit packets at a rate of 2000 packets per second.
- ***Sender -> Ingress*:** occurs due to the increased size of the generation and therefore there is a larger number of elements to buffer.
- ***Ingress -> Egress*:** happens because it is in the egress pipeline that the most intensive computation occurs (i.e. the coding/recoding process). Therefore, the slow processing of packets at the egress causes the packets in the queue to drop.

By mapping the losses to each section we identify how our core modules (i.e., buffering and coding) suffer from the different configurations. Higher generation sizes tend to have a more negative impact on the buffer module due to the higher number of symbols to buffer. Recoding, even with small configuration sizes, has a drastic impact on the coding module due to a high number of multiplications performed.

5.3 Summary

In this chapter, we described our testing environment and the developed testing tools in Sec. 5.1. Then, we presented the evaluation of our work in Sec. 5.2, starting with an analysis of the program itself, followed by a demonstration of its correct functionality. Lastly, we evaluated the performance of the two multiplication algorithms implemented, evaluated the impact of the coding parameters and the cost of recoding, and investigated the root cause of packet loss.

Chapter 6

Conclusion & Future Work

In this thesis, we designed and implemented a switch capable of performing Random Linear Network Coding within the data plane of a switch. This was made possible through the use of a high-level language for switch programming, the P4 language. Our solution improves over previous work [5] by designing and implementing a packet format defined specifically for RLNC, by dynamically performing coding or recoding depending on the incoming packet type, by exploring the trade-offs of different algorithms for multiplication in finite fields, and lastly by studying and evaluating the overall impact of the coding parameters on the performance of the network coding switch, as well as the cost of the recoding operation.

We studied the state-of-the-art regarding the field of Random Linear Network Coding and settled with the most fleshed out and well defined technique: a generation-based RLNC protocol where encoding and decoding operations are performed over generations. More precisely, we comply with a Standard RLNC behavior where the switch waits for all the symbols belonging to the same generation to be received before starting coding. However, even with these well studied definitions, the implementation of such a protocol was not trivial to implement in the data plane due to the intricacies and limitations of the P4 language.

For instance, buffering was not a trivial challenge to solve. The implementation was done through a single register partitioned among generations. Yet, we still had to resort to additional registers to handle the management of the indexation of the generations. This poses the question of whether or not P4-programmable architectures should expose anything more specifically related to the buffering mechanisms of packets in the switch.

Another main obstacle was the absence of the support of cycles in the P4 language. Operations such as buffering/loading symbols to/from registers, performing multiplications over all the elements, would benefit from a loop construct. It would not only facilitate the implementation but would also improve, by a large margin, code readability. We resorted to a P4 code generator to solve this issue. However, this makes it less practical to re-use the code and make it more convoluted to write new code.

Throughout this work, we achieved payload manipulation by treating it as if it were a packet header. However, this only worked because we specifically dealt with small payload sizes that are usually much smaller than the typical packet payload that can be found in real network scenarios. A solution to this issue would be to have some pre-processing of the packets that would split them into smaller fragments before being subjected to network coding.

Given the presented results, in future work we plan to focus on improving the throughput of our proposed solution, starting with the exploration of different coding techniques. The increased number

of multiplications is a direct consequence of using Standard RLNC, where the linear combinations are performed over every symbol of a generation. This proved to cause a major impact on the throughput of the RLNC-Switch. However, the throughput can be improved upon by introducing sparse coding techniques, where not all coding symbols are combined without causing any negative impact. Thus, by employing such a technique, it is possible to reduce the number of multiplications. Moreover, we plan to investigate a seed-based approach to RLNC, where instead of carrying the full coding vector in the packet header, a small random seed can be sent to generate the coding vector. Thus, we reduce the overhead caused by carrying all the coefficients of every symbol in the packet header, with the downside of losing the ability to recode. Nevertheless, the evaluation confirmed that the recoding operation can be very costly. Therefore, studying the advantages and disadvantages of trading the ability to recode for a seed-based approach may be an effort worth pursuing as future work.

Exploring other P4 targets, namely in hardware, such as the Barefoot Tofino switch, could also help us understand better the limitations of our design. The evaluation was entirely performed on the reference P4 software switch, the Bmv2, whose performance entirely depends on the CPU of the machine that is running the switch. A question that remains is therefore if our solution is capable of processing packets at line-rate.

Bibliography

[1] Diogo Figueiredo Pinto. Network coding data planes with programmable switches. 2017.

[2] Muriel Medard. *Network coding: fundamentals and applications*. Academic Press, 2012.

[3] X. Yin, Y. Wang, Z. Li, X. Wang, J. Zhao, and X. Xue. Bounding the advantage of multicast network coding in general network models. *IEEE Transactions on Communications*, 62(3):1023–1032, March 2014.

[4] R. Ahlswede, Ning Cai, S. . R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, July 2000.

[5] 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.

[6] T. Ho, M. Medard, R. Koetter, D.r. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[7] Janus Heide. Random linear network coding (rlnc): Background and practical considerations. internet-draft. ietf secretariat. 2019.

[8] P4 software switch. `https://github.com/p4lang/behavioral-model`, Last accessed on 2018-12-18.

[9] Diogo Gonçalves, Salvatore Signorello, Fernando Ramos, and Muriel Médard. Random linear network coding on programmable switches. *EuroP4 Workshop*, 2019.

[10] R. Ahlswede, Ning Cai, S. Y.R. Li, and R. W. Yeung. Network information flow. *IEEE Trans. Inf. Theor.*, 46(4):1204–1216, September 2006.

[11] T. Ho, R. Koetter, M. Medard, D.r. Karger, and M. Effros. The benefits of coding over routing in a randomized setting. *IEEE International Symposium on Information Theory, 2003. Proceedings.*, 2003.

[12] Wendie Wang, Guozhu Jia, and Menhang Wei. Efficient multicast in wireless networks using network coding. *2015 IEEE International Conference on Communication Problem-Solving (ICCP)*, pages 126–129, 2015.

[13] D. Silva and F. R. Kschischang. Universal secure network coding via rank-metric codes. *IEEE Trans. Inf. Theor.*, 57(2):1124–1135, February 2011.

[14] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air. *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM 06*, 2006.

[15] Zimu Liu, Chuan Wu, Baochun Li, and Shuqiao Zhao. Uusee: Large-scale operational on-demand streaming with random network coding. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

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

[17] Kodo - introduction to network coding. `http://docs.steinwurf.com/nc_intro.html`, Last accessed on 2018-10-11.

[18] Lester R Ford Jr and Delbert R Fulkerson. Maximal flow through a network. 8:399–404, 01 1956.

[19] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. A complete treatment of software implementations of finite field arithmetic for erasure coding applications. 2014.

[20] Leonard Eugene Dickson. *Linear groups: With an exposition of the Galois field theory*. Courier Corporation, 2003.

[21] Diego Kreutz, Fernando M. V. 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.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[23] 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. volume 43, pages 99–110, 08 2013.

[24] The P4 Language Consortium. `https://p4.org/p4-spec/docs/P4-16-v1.0-spec.html`, Last accessed on 2018-10-12.

[25] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[26] Rob Sherwood, Glen Gibb, Kok kiong Yap, Martin Casado, Nick Mckeown, and Guru Parulkar. Can the production network be the testbed. In *In USENIX Symposium on Operating Systems Design and Implementation (OSDI*, 2010.

[27] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J Jackson, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, volume 14, pages 203–216, 2014.

[28] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: performance, isolation, and velocity at scale in cloud network virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 373–387, 2018.

[29] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[30] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28. ACM, 2017.

[31] Cheng Huang and Lihao Xu. Fast software implementations of finite field operations * (extended abstract). 2003.

[32] Kevin Greenan, Ethan Miller, and Thomas Schwarz. Analysis and construction of galois fields for efficient storage reliability. 09 2007.

[33] Annabell Kuldmaa. Efficient multiplication in binary fields. 2015.

[34] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[35] Product brief tofino page | barefoot. `https://barefootnetworks.com/products/brief-tofino/`, Last accessed on 2018-12-18.

[36] Navigators' quinta description. `https://navigators.di.fc.ul.pt/wiki/Quinta`. Accessed: 2019-07-24.

[37] Philippe Biondi. Scapy documentation. `https://scapy.readthedocs.io/en/latest/`, Last accessed on 2019-31-07.

[38] Emin Martinian. Python package: Pyfinite. `https://github.com/emin63/pyfinite`, Last accessed on 2019-31-07.

[39] S. Jaggi, P. Sanders, P.a. Chou, M. Effros, S. Egner, K. Jain, and L.m.g.m. Tolhuizen. Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982, 2005.

[40] S.-Y.r. Li, R.w. Yeung, and Ning Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[41] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, 11(5):782–795, October 2003.

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

[43] Kodo - frequently asked questions. http://docs.steinwurf.com/faq.html#what-is-a-code, Last accessed on 2018-10-16.

[44] The P4 Language Consortium. P4 adoption continues to grow rapidly. https://p4.org/technical-steering-committee/p4-adoption-continues-to-grow-rapidly.html, Last accessed on 2018-10-16.

[45] Kodo payload format. https://github.com/steinwurf/kodo-rlnc/blob/master/src/kodo_rlnc/detail/payload_writer.hpp, Last accessed on 2018-12-18.

[46] Karamjeet Kaur, Japinder Singh, and Navtej Singh Ghumman. Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42, 2014.

[47] Yao Li, Emina Soljanin, and Predrag Spasojevic. Collecting coded coupons over overlapping generations. In *Network Coding (NetCod), 2010 IEEE International Symposium on*, pages 1–6. IEEE, 2010.

[48] 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.

[49] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept 2014.

[50] Europ4 workshop. https://p4.org/events/2019-09-23-euro-p4-workshop/, Last accessed on 2019-23-09.

[51] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru M. Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: make your virtual sdns programmable. In *HotSDN*, 2014.

[52] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[53] Max Alaluna, Eric Vial, Nuno Neves, and Fernando M. V. Ramos. Secure and dependable multi-cloud network virtualization. In *Proceedings of the 1st International Workshop on Security and*

Dependability of Multi-Domain Infrastructures, XDOMO'17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.