

**RANDOM LINEAR NETWORK CODING OVER SOFTWARE
DEFINED NETWORKS**

BY

AHMED ALI MOHAMED HASSAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER NETWORKS

May, 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN- 31261, SAUDI ARABIA
DEANSHIP OF GRADUATE STUDIES

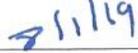
This Thesis, written by **AHMED ALI MOHAMED HASSAN** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER NETWORKS**.



Dr. Ahmad Al-Mulhem
Department Chairman

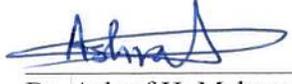


Dr. Salam A. Zummo
Dean of Graduate Studies

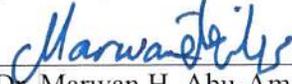


Date





Dr. Ashraf H. Mahmoud
(Advisor)



Dr. Marwan H. Abu-Amara
(Member)



Dr. Tarek Sheltami
(Member)

© Ahmed Ali Mohammed Hassan
2018

*Dedicated to my
Mother, Father, siblings and their sacrifices
and to my small family
Abeer and Anne*

ACKNOWLEDGMENTS

All praise is to Allah, the Almighty alone. May the Peace and Blessings of Allah be upon the Messenger of Allah (salallahualewasalam), his family, and his companions (radhiiallahoanhum).

I am grateful to the King Fahd University of Petroleum & Minerals for providing a great environment for research and academics. I wish to extend my gratitude to my thesis adviser Dr. Ashraf Hassan Mahmoud for his continuous support, patience, and much needed encouragement. I am also thankful to my thesis committee Dr. Marwan Abu Amara and Dr. Tarek Sheltami for their time and useful comments.

My mother (Aiesha) is always with me in my thoughts and I live with her encouragement and kindness. All good I do goes to my mother. For all my studies. I am also thankful to my brother, sisters and daughter (Anne) for their encouragement and support. I am thankful to my wife (Abeer) for her patience and help during long hours of studies and research, she equally shares this work.

I am very thankful to my friends at KFUPM for making my master full of enjoyment. I am indebted to all the people I meet here at KFUPM.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	III
TABLE OF CONTENTS	IV
LIST OF TABLES	VII
LIST OF FIGURES	VIII
LIST OF ABBREVIATIONS	X
ABSTRACT	XI
ملخص الرسالة	XII
CHAPTER 1 INTRODUCTION	1
1.1 Background	1
1.1.1 Notations and Definitions.....	1
1.2 Network Coding Theory	4
1.2.1 Linear Network Coding (LNC)	6
1.2.2 Random Linear Network Coding (RLNC)	8
1.3 Software Defined Networks (SDN)	9
1.4 Problem Statement	14
1.5 Objectives	15
1.6 Methodology	15
1.6.1 Network coding aware routing module.....	15
1.6.2 Network management module	16
1.7 Implementation Requirements	16
1.7.1 Simulation Environment	16
1.8 Deliverables and Outcomes	18
CHAPTER 2 LITERATURE REVIEW	19
2.1 LITERATURE REVIEW	19
CHAPTER 3 NC-SDN EXTENDED FRAMEWORK	32

3.1 Overview	32
3.2 Data plane NC integration	32
3.2.1 OpenFlow Module	33
3.2.2 OpenFlow Network Coding Messages	36
3.2.3 Network coding Flow Modification Messages	38
3.2.4 OpenFlow Matching	40
3.2.5 OpenFlow Extensible Matching	42
3.2.6 OpenFlow Instructions	44
3.2.7 Network coded packet structure	46
3.2.8 DPDK buffers management module	47
3.2.9 Random Linear Network Coding Implementation	51
3.3 Control Plane NC architecture	58
3.3.1 Flow Entry Management module	58
3.3.2 NC generation management module	58
3.3.3 Network management Module	59
3.3.4 Multicast-Multipath Network Coding Aware Routing Module	61
 CHAPTER 4 RLNC-SDN FRAMEWORK VALIDATION	 62
4.1 Overview	62
4.2 Centrality Computation	62
4.2.1 Degree Centrality	62
4.3 Shortest Path-tree Computation	63
4.4 Flow Computation	66
4.5 Packet Encoding and Decoding Process Verification	68
 CHAPTER 5 EXPERIMENTAL SETUP AND IMPLEMENTATION TOOLS.....	 74
5.1 Experiment Environment and Tools.....	74
5.1.1 Mininet	74
5.1.2 Wireshark	75
5.1.3 RYU Controller	76
5.1.4 OpenvSwitch.....	80
5.1.5 RLNC Libraries.....	82
5.2 Performance metrics	82
5.2.1 Throughput	82
5.2.2 Delay /Latency	82
5.2.3 CPU Utilization	82
5.3 The Experimental Steps	83
5.4 SDN Multi-hop Reference Scenario	84
 CHAPTER 6 SDN-NC EXPERIMENTS AND RESULTS.....	 85

6.1 Butterfly Scenario	85
6.1.1 Throughput	87
6.1.2 Delay/Latency	90
6.1.3 CPU Utilization / Computation Power	92
6.2 Multicast Fat-tree scenario	95
6.2.1 Throughput	97
6.2.2 Delay/Latency	100
6.2.3 CPU Utilization / Computation Power	102
CHAPTER 7 SUMMARY AND FUTURE WORK	105
7.1 Summary	105
7.2 Thesis Accomplishments	105
7.3 Future Work	106
REFERENCES	107
APPENDICES	109
A. OpenVswitch RLNC modifications	109
B. Mininet Topology scripts	128
C. Ryu Controller Applications	131
VITAE	135

LIST OF TABLES

Table 2.1	Summary of the literature NC-SDN frameworks	31
Table 3.1	NCoS framework OpenFlow matching entries.....	41
Table 3.2	NC-SDN implementation OpenFlow matching entries	41
Table 6.1	Mininet SDN-RLNC butterfly configuration parameters	86
Table 6.2	Mininet SDN-RLNC Fat-tree configuration parameters	97

LIST OF FIGURES

Figure 1.1	Network graph types.....	2
Figure 1.2	Network coding for the butterfly network.....	6
Figure 1.3	Butterfly example of linear network coding.....	7
Figure 1.4	Simplified view of Software Defined Network Architecture.....	11
Figure 1.5	Butterfly SDN network scenario prototype.....	17
Figure 2.1	Overview of OpenFlow Controller proposal for IP multicast networking ...	26
Figure 2.2	NCoS framework.....	29
Figure 3.1	The proposed SDN-NC framework architecture.....	32
Figure 3.2	OpenFlow header structure.....	34
Figure 3.3	OpenFlow experimenter message structure.....	34
Figure 3.4	Controller connection state machine	35
Figure 3.5	Switch connection state machine.....	35
Figure 3.6	OpenFlow connection establishment phase.....	36
Figure 3.7	OpenFlow features discovery phase.....	37
Figure 3.8	OpenFlow FeatureRes message structure.....	38
Figure 3.9	OpenFlow Encode action message structure.....	39
Figure 3.10	OpenFlow Decode action message structure.....	40
Figure 3.11	OpenFlow 1.0 fixed matching structure	42
Figure 3.12	OpenFlow Extensible Matching structure	43
Figure 3.13	OpenFlow TLV structure	45
Figure 3.14	OpenFlow instructions in a TLV payload	45
Figure 3.15	Proposed network coding packet structure.....	47
Figure 3.16	Ethernet frame encapsulation of NC packet.....	47
Figure 3.17	Data Plane Development Kit Architecture	48
Figure 3.18	NC packets buffering in DPDK module.....	50
Figure 3.19	DPDK buffers in a NC decoder node	50
Figure 3.20	RLNC Encoding and Decoding process.....	52
Figure 3.21	SDN Topology discovery mechanism.....	60
Figure 4.1	Butterfly shortest disjoint paths.....	65
Figure 4.2	The computed butterfly path-tree via Dijkstra algorithm.....	65
Figure 4.3	OpenFlow interactions of the statistics module.....	66
Figure 4.4	OpenFlow statistics database.....	67
Figure 4.5	UDP Ethernet frame structure	68
Figure 4.6	RLNC generation of six Ethernet frames	69
Figure 4.7	Original UDP packet that generated via Iperf tool.....	70
Figure 4.8	Encoded packet encapsulated by NC Ethernet frame.....	71
Figure 4.9	RLNC Decoder generation status matrix	72
Figure 4.10	RLNC decoder status matrix of full recovered NC generation	73
Figure 5.1	OpenFlow Feature Response data	79
Figure 5.2	OpenFlow port statistics response data	80
Figure 5.3	SDN multi-hop topology	84
Figure 6.1	Mininet SDN-RLNC butterfly topology	85
Figure 6.2	Butterfly Throughput of three traffic types: UDP, encoded and decoded.....	87
Figure 6.3	Butterfly Throughput versus UDP traffic load.....	90

Figure 6.4	Butterfly coding delay/latency versus UDP traffic load.....	92
Figure 6.5	Butterfly decoding delay/latency vs UDP traffic load	92
Figure 6.6	Butterfly CPU Utilization.....	94
Figure 6.7	Butterfly CPU utilization versus UDP traffic load.....	94
Figure 6.8	Mininet SDN-RLNC Fat-tree Topology	96
Figure 6.9	Fat-tree Throughput of three traffic Types: UDP, encoded and decoded	97
Figure 6.10	Fat-tree topology encoded throughput versus UDP traffic load.....	99
Figure 6.11	Fat-tree topology decoded Throughput versus UDP traffic load	100
Figure 5.12	Fat-tree traffic latency of encoded traffic versus UDP traffic load	101
Figure 6.13	Fat-tree decoded traffic delay versus UDP traffic load	102
Figure 6.14	Fat-tree CPU utilization for encoding nodes s4, s5 and s6.....	103
Figure 6.15	Fat-tree CPU Utilization for Decoding nodes S8,S9,S10,S11,S12 and S13	104

LIST OF ABBREVIATIONS

NC : Network Coding

SDN : Software Defined Networks

IPv4 : Internet Protocol Version 4

LNC : Linear Network Coding

RLNC : Random Linear Network Coding

GF : Galois Field

P2P : Peer to Peer

CNCNS : Centrality-based Network Coding Node Selection

IGMP : Internet Group Management Protocol

OXM : OpenFlow Extensible Matching

ETHTYPE : Ethernet Protocol Type

ToS : Type of Service

AES : Advanced Encryption Standard

TCP : Transport Control Protocol

LLDP : Link Layer Discovery Protocol

DPDK : Data Plane Development Kit

ABSTRACT

Full Name : Ahmed Ali Mohammed Hassan

Thesis Title : Random Linear Coding over Software Defined Networks

Major Field : Computer Engineering

Date of Degree : May 2018

The vertical integration and layered structure of current legacy communication networks such as the Internet has limited the rapid evolution in multicast networks. Thus, Network coding (NC) and Software Defined Networks (SDN) are recent emerging networking concepts with remarkable potentials in enabling a higher network performance via flexible and scalable network architecture. The global view and programmability nature of SDN allows to realize the technical requirements for network coding in wired multicast networks.

In this work, we developed a novel framework for Random Linear Network Coding (RLNC) for an SDN architecture and implement a proof of concept prototype to characterize the performance of NC in terms of throughput, delay and coding/decoding processing time in comparison with conventional routing for different multicast network scenarios. |

ملخص الرسالة

الاسم الكامل: أحمد علي محمد حسن

عنوان الرسالة: ترميز الشبكات العشوائية الخطي على الشبكات المعرفة برمجياً

التخصص: هندسة الحاسوب

تاريخ الدرجة العلمية: مايو 2018

إن الترابط والتكامل الرأسي والتركيبية متعددة الطبقات لشبكات الإتصالات التقليدية مثل شبكة الإنترنت حدثت من التطور المتسارع في شبكات تقنية البث المتعدد. لذلك تقنياتي ترميز الشبكات والشبكات المعرفة برمجياً هي مفاهيم جديدة في تقنيات الشبكات ذات إمكانيات ملحوظة في تطوير وتحسين شبكات البث المتعدد بأداء عالي ضمن معمارية شبكات مرنة وقابلة للتوسع.

الرؤية الشاملة وقابلية البرمجة للشبكات المعرفة برمجياً أتاحت تطويع المتطلبات التقنية لتطبيق ترميز الشبكات في شبكات متعددة البث.

في هذا البحث، قمنا بتطوير نموذج عمل لتشفير الشبكات ذات الترميز الخطي العشوائي ضمن معمارية الشبكات المعرفة برمجياً وتنفيذ نموذج تجريبي لدراسة الأداء للترميز الشبكي ضمن معايير الإنتاجية ومعدل التأخير وعمليات الترميز وفك الترميز والوقت المستهلك لذلك ومقارنة النتائج بشبكات البث المتعدد التقليدية لأنواع مختلفة من شبكات البث المتعدد.

CHAPTER 1

INTRODUCTION

1.1 Background

In this section, basic concepts and laws in information and graph theory are defined to develop a clear understanding of network coding theory.

1.1.1 Notations and Definitions

- **Network system**

A system consists of a set of information sources and communication nodes connected by channels or links to exchange information between the nodes such as computer networks, telephone networks.

- **Network graph**

A finite connected and directed graph or model is denoted as $G = (V, E)$ where V is a set of nodes or vertices and E is a set of edges linking these nodes [1]. An edge in E also refers to a communication channel. The network graph has three types of nodes:

- A source node s is a node without any incoming edges.
- A sink node t is a node without any outgoing edges.
- An intermediate or non-source node that is linked with one or more incoming and outgoing edges.

When the network does not have any directed cycles, or loops it is called an acyclic network otherwise it is called a cyclic network. If it has only a single-source node it is called a single-source network, and if it contains multiple sources, then it is referred as a multi-source network. Figure 1.1 shows the components and types of network graphs. In this research, we will focus only on the directed-acyclic graph networks type where there is no more than one path connects two nodes in the same direction.

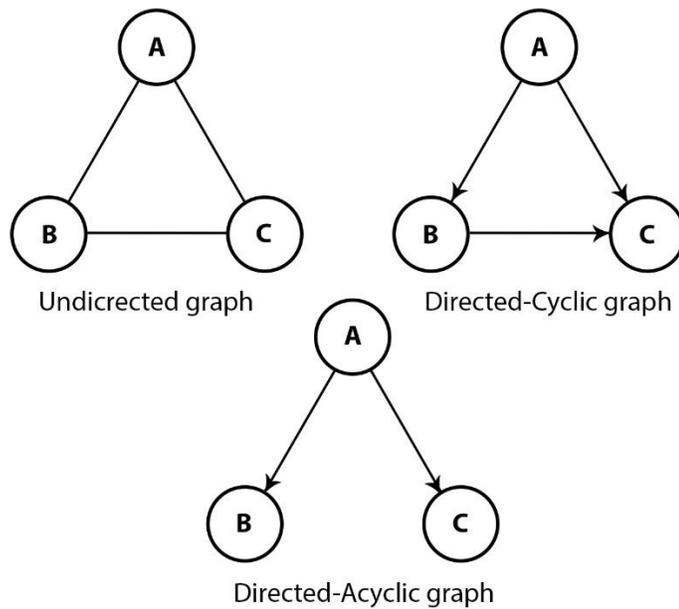


Figure 0.1 Network graph types

- **Multicast Network**

A multicast network is a network where a source node wishes to communicate a message to a set of destinations or sink nodes. Multicasting is employed in applications such as media streaming, distributed storage systems or any other multipoint communications.

- **Channel capacity (C)**

For a channel or edge E , the maximum number of information symbols taken from a finite alphabet that can be sent on the channel per unit time is called the channel capacity or constraint rate.

- **Network flow (F)**

Network flow is the rate of information transmission on the channel going into or outgoing of a node.

- **Law of commodity flow**

The total volume of the out flow from a non-source node cannot exceed the total volume of the flow entering that node.

- **Law of information flow**

The content of any information flowing out of a set of non-source nodes can be derived from the accumulated information that flows into that set of nodes.

- **Network capacity**

The network capacity is the maximum capacity of a network path to convey data from a source node to a sink node in the network.

- **$s - t$ cut**

An $s - t$ cut is a partition of graph vertices V with respect to two distinguished nodes s and t that belongs to two different subsets S and \bar{S} where the two subsets satisfy the property that $s \in S$ and $t \in \bar{S} = V - S$

- **Maximum flow problem**

In a capacitated network, i.e. a network with capacity constraints, we intend to send as much flow as possible between a source node s and a sink node t , without exceeding the capacity of any channel or link.

- **Max-flow Min-Cut Theorem**

The maximum value of the flow from a source node s to a sink node t in a capacitated network equals the minimum capacity among all $s - t$ cuts.

1.2 Network Coding Theory

Traditional information delivery via a network can be defined as an exchange of data pieces, without the ability of combining or mixing what was sent as defined previously in the commodity flow law. In 2000, Ahlswede *et. al* [2] changed this perspective by introducing the concept of information flow that allows the combine information to increase the capacity of a network over the limit achieved by conventional store-and-forward routing.

Traditional coding techniques are referred to as source-based coding, where only source nodes encode packets. In network coding, non-source nodes are required to encode the

input packets together before sending them out in order to achieve multicast delivery at the maximum possible data transfer rate. Therefore, we can define network coding in general as coding at a node for a packet while in transit in the network [3].

To study network coding evolution, we should simplify the communication network model by introducing a number of required assumptions as following:

- A generated message at the source node s consists of a number of data units, each is presented by a symbol that belongs to a certain base or finite field.

Finite field (F) is defined as a finite set of elements for which the operations of commutative multiplication, addition, subtraction and division performed within its elements and result in another element of the same set [4].

- A network model is graphed as an acyclic directed single-source multicast network. The most popular studied example model is the butterfly network graph [2] that is depicted in Figure 1.2.

- The channels through the network are assumed to be lossless and reliable to deliver coded packets to the sink nodes.

- All capacities are non-negative integers and the network does not contain parallel links or channels.

In a typical multicast scenario, we use the butterfly network graph shown in Figure 1.2 to illustrate a simple example of the network coding technique. In this example, the source node, node 1, intends to deliver a stream of messages $M1$ and $M2$ to both destination nodes: node 6 and node 7. Assuming all links have a capacity of one message per unit time and middle nodes: 2, 3, 4 and 5 only forward the messages they receive. It is easy to see that

the middle link or edge (4, 5) is the main bottleneck, as it cannot forward more than one message per transmission or unit time. The max-flow min-cut theorem, as mentioned early, predicts that the upper bound of multicast capacity for this network is two messages per transmission or unit time. If we use a simple encoding operation at the middle or relay nodes such as XORing the two messages at node 4. Then, both destination nodes 6 and 7 can receive $M_1 \oplus M_2$ in one-unit time or a single transmission. So, the upper-bound capacity has been achieved. However, the network coding idea can be generalized to achieve the multicast capacity for arbitrary multicast networks with more generic network codes [5].

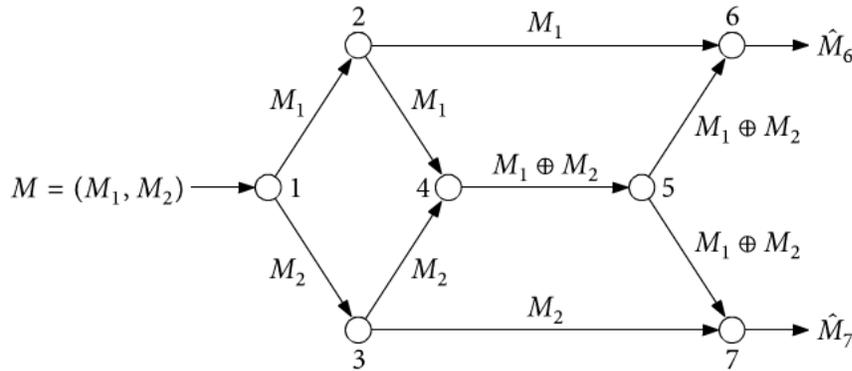


Figure 0.2 Network coding for the butterfly network

1.2.1 Linear Network Coding (LNC)

A linear network coding linearly combines information and coefficients or code words chosen from a finite field (F). For example, $F^2 = GF(2)$, where GF refers to the Galois field notation, and GF(2) refers to the case of data units being either bit 0 or 1.

For example, let m_i be the native message x symbols for $i = 0, 1, 2, \dots, N$ where N is the total number of symbols and let $C_i = (c_{i,1}, c_{i,2}, c_{i,3}, \dots, c_{i,N})$ donates the linear local coefficients or coding vectors $C_i \in F^\omega$, where ω is the symbol size. Furthermore, let y_i for $i = 0, 1, 2, \dots, M$ represents the received coded symbols at a sink node t and M is the total

number of the received coded symbols. Then, a linear system is generated at the sink node t , where $Y = Cx$ as shown in equation (1.1).

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} = \begin{pmatrix} c_{1,1} & \cdots & c_{1,N} \\ \vdots & \ddots & \vdots \\ c_{M,1} & \cdots & c_{M,N} \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_N \end{pmatrix} \quad (1.1)$$

Note that $M \geq N$. In order to retrieve the original symbols, the sink node needs to recognize the coding vectors of the coded symbols that have been received. Hence, it is required to embed the coding vectors with coded message data units in order to deliver these coding vectors to the receiver node.

Figure 1.3 shows a more detailed example of linear coding in the butterfly network graph where nodes S and R correspond to the source node and the sink node, respectively.

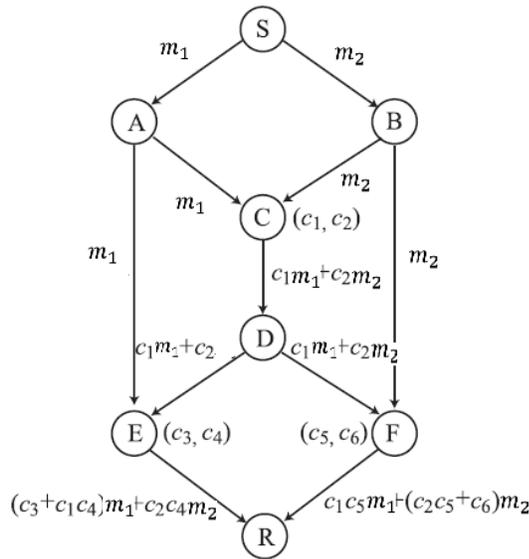


Figure 0.3 Butterfly example of linear network coding

Each of the intermediate nodes C , E and F has two input links and one output link, and coefficients (C_1, C_2) , (C_3, C_4) , and (C_5, C_6) are assigned to output links of node C , E , and F , respectively. Intermediate nodes A , B , and D only forward received packets without any coding. Source node S transmits symbols m_1 and m_2 to its two output links (S, A) and

(S, B) , respectively, and sink node R receives two coded symbols, $y_1 = (C_3 + C_1C_4)m_1 + C_2C_4m_2$ and $y_2 = C_1C_5m_1 + (C_2C_5 + C_6)m_2$. Therefore, the system of linear equations for native symbols m_1 and m_2 will be generated as following:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} C_3 + C_1C_4 & C_2C_4 \\ C_1C_5 & C_2C_5 + C_6 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \end{pmatrix}. \quad (1.2)$$

1.2.2 Random Linear Network Coding (RLNC)

In 2003 Ho *et al.* [6], defined a random linear coding method for multicast networks, where the nodes send linear combinations of the incoming information on the outgoing channels, using independent and randomly chosen code coefficients from some finite fields. Since each node can choose its own encoding coefficients independently of the other network nodes, the network coding can be more suitable for network topologies with unknown or dynamic structure.

This approach makes network coding more suitable for multicast networks with unknown or dynamic topologies. In addition, the decoding failure probability in receiver nodes can be arbitrarily reduced by increasing the size of the finite field that generates the random codes i.e. the decoding failure probability decreases exponentially with the increase of the number of bits in the codewords or symbols.

RLNC enhances network coding efficiency via recoding process in the intermediate nodes. Recoding enables an intermediate node to re-encode the received coded packets on its incoming edges by generating new coded packets with different local coding vectors or codewords. This will make coded packets are less likely to be linearly dependent and reduce the overall delay of decoding at the receiver nodes.

The RLNC can be categorized into two types of coding:

- **Systematic RLNC coding**

The source sends first all original or not coded packets along the way to destinations, then it sends the coded packet to destinations in order to correct corruptions and recover any losses in the not coded packets [7].

- **Non-systematic coding**

The source transmits only the coded packets to destinations without sending the original packets in the beginning of the transmission session [7].

1.3 Software Defined Networks (SDN)

In conventional data network infrastructure, the control plane defines the protocols and software components that take forwarding decisions, where these protocols and software components are bundled with the data plane that executes packet forwarding.

In addition, the traditional IP network structure is highly decentralized and the vertical integration and layering in today's networks makes it extremely difficult to evolve and has introduced many limitations in networking flexibility and scalability [8].

Software Defined Networks (SDN) [9] is an emerging data communication paradigm that separates the control plane from the data plane. This separation provides more flexible, programmable, vendor-independent, cost efficient, and innovative network architecture.

The main characteristics of SDN is the centralization and network programmability of the control plane. In this paradigm, the logically centralized controller is the entity responsible for the control logic, administration, and monitoring the network processes and operations.

The data plane is abstracted to impose forwarding logic only via compliant software or hardware forwarding nodes.

In general, the SDN architecture consists of four main innovations as shown in figure 1.4:

- 1- The control plane and data plane are decoupled.
- 2- Forwarding actions or instructions are defined on a flow-basis not on a destination-basis.
- 3- Control logic is defined as a separate entity called the SDN controller. This controller takes the responsibility of installing the control commands, flow routes in the forwarding devices and gathering information about the forwarding plane elements e.g. network nodes and links, to offer a global and real-time network view to upper network applications.
- 4- The network is programmable, where software programs running on top of control plane can interact with the underlying data plane devices via standardized programmable interfaces such as OpenFlow [10].

The separation of data and control signaling is not a new concept as it originated in the telephone networks where the Network Control Point (NCP) [11] was introduced by AT&T to enhance control and management of telephony networks.

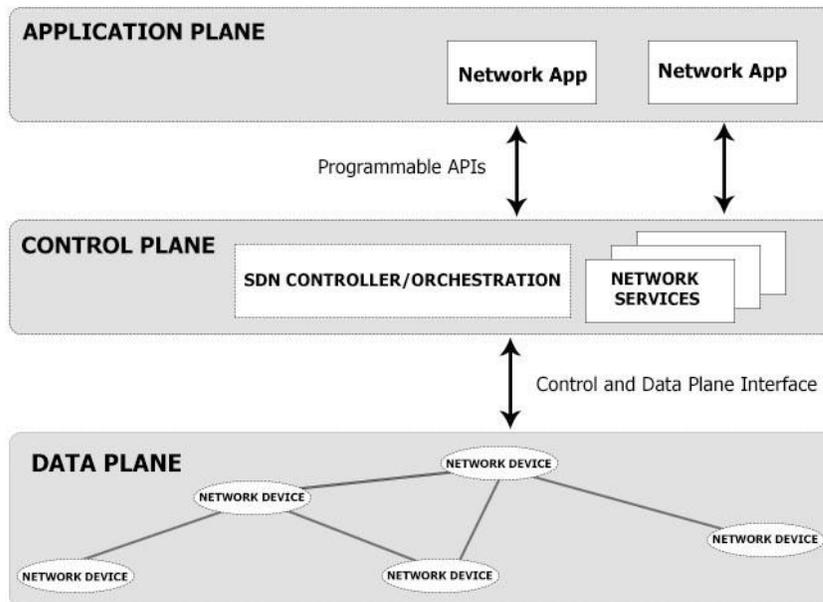


Figure 0.4 Simplified view of Software Defined Network Architecture

OpenFlow [10] protocol is the concrete realization of the SDN approach which is a standardized protocol to assure configuration and communication compatibility between SDN control entities and separate forwarding hardware. It allows researchers to re-engineer the network traffic and test new protocols in existing networks without disrupting upper network applications.

The OpenFlow enabled device is based on a pipeline of flow tables where each entry has three elements:

- 1- A rule to be matched.
- 2- An action to be executed on matching packets where it can be forwarding a matching packet or modifying it. Actions maybe accumulated or applied immediately to the packet.
- 3- Counters to keep statistics of matching packets.

While there are significant benefits of SDN in the evolution of data networking, there are many challenges and research areas that have been widely raised. These challenges include, but not limited to, the following topics:

- **Quality of Service (QoS)**

Quality of Service is the ability to provide a service that satisfies parameters like the required bandwidth, minimum delay, packet loss or jitter to guarantee a certain level of performance.

There are many contributions specific to support QoS in an SDN architecture. Examples include OpenQoS and OpenQFlow [12].

- **Congestion aware routing**

One of the evident techniques in congestion-aware routing is load balancing. It is utilized to enhance SDN network services availability, scalability and to lead to minimal response time for the upper layer applications. One of the well-known load balancing methods is the Equal Cost Multipath (ECMP) [13], which is a routing mechanism that calculates the cost of multiple paths and distributes the traffic over them based on the computed cost. Valiant Load Balancing (VLB) is another load balancing strategy to forward incoming flow to a respective destination by selecting a switch randomly along the way to that destination. Both mechanisms ECMP and VLB are applied mostly in data centers [13].

- **Scalability**

SDN scalability issues can be addressed at three different levels [12]:

- 1- The number of switches that the logic controller can support.

2- The flow table memory capacity inside network forwarding devices.

3- The heterogeneity of SDN switches and how the controller is capable to handle them in multiple sparse locations.

- **Security and Dependability**

Security is a major threat in deployed SDN networks especially in datacenters.

One of the most important security challenges is how to protect the control plane communications with the underlying data plane. Some of security frameworks are developed to mitigate specific SDN network threats such as distributed denial-of-service attack (DDoS), Intrusions, network policy violations...etc. Fresco and netFuse are examples of SDN security modules [8].

The extendibility of SDN/OpenFlow is the key feature that encourages researchers to apply new networking approaches or solutions and investigate the possibilities of implementing theoretical techniques without creating a fundamental impact on the current network protocols and infrastructure or effecting its heterogeneity.

1.4 Problem Statement

The extensive realization of network coding on existing SDN network architectures is still far from wide spreading especially in multicast wired networks. The following challenges are the focus of the thesis elements to be tackled in adopting and utilizing network coding to maximize the multicast throughput and reduce the traffic overhead. To identify these challenges, we can classify them into three significant challenges:

- **Inter/intra network coding aware routing**

The paths that guide data flows in the network should guarantee the innovativeness and independent property of encoded packets along the way to their destinations. So we need to find an optimal routing protocol to increase network throughput with minimum bandwidth cost.

- **Coding and decoding complexity**

Encoding process in intermediate nodes incurs a delay and increases the overall network complexity. Encoding node has both forwarding and coding capabilities. Therefore, it is more expensive in comparison to regular forwarding node in term of resources such as memory and computation power. To reduce this complexity, we need to minimize the number of encoding nodes without a degradation in the network coding performance [14].

- **Flexibility and backward compatibility**

The variety of coding schemes and disparity in their features add another dimension of complexity where they lead to an incremental deployment problem and compatibility issue with current TCP/IP protocols stack [15]. The flexibility is substantial to realize network coding in current network structures.

1.5 Objectives

This work intends to extend the design and the implementation of proposed frameworks [16], [17] using the same RLNC encoding scheme as in [18] with consideration for more complex topologies not only the simplified linear multi-hop ones. Our work expects to meet the following objectives:

- 1- Identify an optimal network coding aware multipath routing that ensures the efficiency of encoding and decoding with minimum computation overhead benefiting from the global view of SDN.
- 2- Minimizing the number of enabled network coding nodes in the multicast network.
- 3- Evaluating the performance in terms of throughput and delay for different network topologies.

1.6 Methodology

The centralized control of SDN network architecture and the flexibility of network functionalities are the main key features we use to tackle the network coding challenges by exploiting the programmability of the logically central network controller. In our solution, we will extend the design of proposed architecture in [17] by applying different techniques in controller function modules in attempt to increase NC throughput and decrease the computation overhead.

In following we will discuss the proposed techniques for function modules of interest.

1.6.1 Network coding aware routing module

In Multicast routing function, we will try to consider the following guidelines:

- Sink node should receive the sufficient number of encoded packets so it can decode the original multicast packets.
- Paths from source to each sink node should guarantee the independency of encoded packets to insure the delivered packets are decodable at each sink node.
- Compute the admitted flows and current network bandwidth.

In the proposed framework [17], it suggests max-flow iterative algorithm as a multicast routing to find the shortest disjoint paths. In our solution, we suggest using network coding constrained routing (NC-CBR) algorithm that proposed in [19] as our NC aware routing algorithm.

1.6.2 Network management module

To reduce the encoding/decoding complexity issue we will try to minimize the number of intermediate encoding nodes without a negative impact on the gained NC multicast throughput. SDN controller will compute CNCNS [20] algorithm to select which nodes are suitable to do the forwarding only or encoding and decoding along with forwarding function.

1.7 Implementation Requirements

1.7.1 Simulation Environment

For experimental setup and evaluation, we will use Mininet emulator which is known as an open source SDN network systems emulator. Mininet can be used to emulate all network and switch elements such as controllers, switches, and hosts.

The study will simulate different multicast network topologies and collect data for performance analysis and evaluation.

Beside that we will emulate SDN elements, where these elements are controllers, switches, hosts and links.

Figure 1.5 shows a prototype of butterfly multicast as an example of RLNC-SDN multicast network where it has two sources, five intermediate nodes, and two destinations with an SDN controller that interacts with the intermediate nodes via OpenFlow protocol.

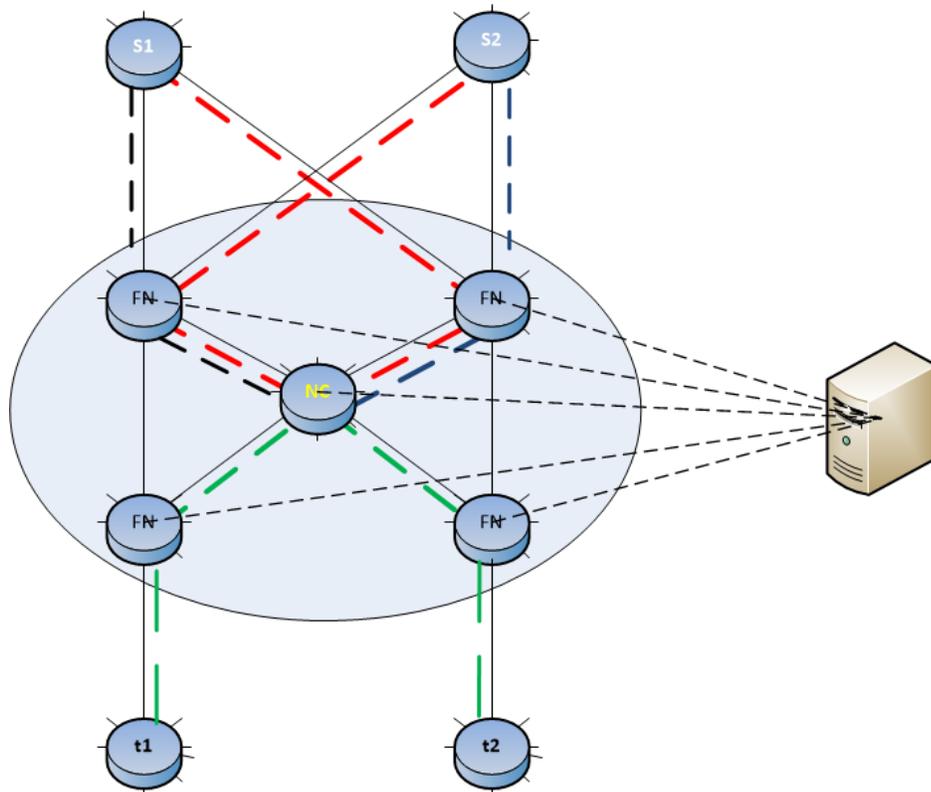


Figure 1.5 Butterfly SDN network scenario prototype

1.8 Deliverables and Outcomes

- A detailed literature survey of NC-SDN frameworks and prototypes.
- A design of efficient prototype of multicast-aware RLNC-SDN controller and OpenFlow forwarders that support RLNC functions.
- Performance analysis of the designed prototype for different multicast network scenarios in comparison with previous frameworks in terms of end-to-end delay, network throughput, and the computation overhead.

|

CHAPTER 2

LITERATURE REVIEW

2.1 LITERATURE REVIEW

In [21] Li, Yeung, and Cai study the concept of linear network coding and prove theoretically that linear coding can achieve the maximum flow in a multicast network from the source to each receiving node. The notation of Linear-Code Multicast (LCM) is defined as an abstract algebraic description of a linear code on a data network. The authors explain how their generic LCM can sufficiently achieve the max-flow bound, generalize it to arbitrary set of max-flow values, define a construction scheme for the generic LCM in memory and memoryless acyclic multicast networks, and define the transmission scheme that associated with it.

In [22] Zhu, Li, and Guo take the advantage of unique characteristics of multicast in application layer and attach network coding capabilities with it to improve the multicast end-to-end throughput.

They propose a two-stage distributed algorithm. The first stage builds a 2-redundant acyclic multicast graph as an overlay multicast topology by the consolidation of two basic graphs called the Rudimentary graph and the Rudimentary tree. The constructed multicast graph is constrained by the number of intermediate nodes and the degree of each node. This

degree refers to the number of ingress and egress edges of a node. In addition, the delay and bandwidth are used as weight metrics for adding or dropping a link from the constructed graph.

The second stage obtains and distributes the linear codes for the 2-redundant multicast graph in two phases: (1) assignment and (2) dissemination of linear codes. This algorithm has introduced a balance between links' costs and the selection of paths to achieve a higher throughput.

Their work has been supported by analytical and simulation results that show a significant improvement in the multicast session throughput in comparison with two conventional multicast protocols: Narada and Distance Vector Multicast Routing Protocol (DVMRP). In terms of end-to-end delay, stress, and resource usage versus the number of receivers, it performs slightly worse than the previously mentioned conventional multicast protocols. The focus of this study was only on the single-source multicast scenarios and the algorithm limits the number of incoming edges to two links at most for each node in the constructed graph to simplify the distribution of linear codes and ensure the data recovery at the receiving nodes.

In our work, we will focus on multicast scenarios with dynamic number of incoming and outgoing edges and randomize the selection of linear codes.

In [23] Wang and Li, design a live peer-to-peer protocol (P2P) called R^2 that utilizes network coding approach to improve the performance of live streaming in P2P networks.

In R^2 protocol, the live stream is divided into data pieces and each piece is divided into fixed-sized blocks. When a seed or source selects and encodes randomly data pieces and explicitly sends encoded blocks to a downstream peer this process is defined as random

push mechanism where no request is made by the downstream peer. The randomized selection is restricted to a priority region that refers to an urgent range in the stream playback time. The seed requests a feedback of missing pieces on its downstream peers. In classical P2P protocols, a buffer map is interchanged periodically between peers which states the availability of each data piece in the playback buffer. The design of R^2 uses the same bitmap buffer strategy to obtain the knowledge of missing pieces but it is exchanged with higher frequency, so the peer sends its buffer map whenever it has played back piece or it has completed downloading it. Random network coding is the cornerstone of R^2 protocol, when a data piece is selected to be encoded by a seed for its downstream peer. The seed chooses an independent and random set of coding coefficients C_i of binary field $GF(2^8)$ for each block b_i to be sent to downstream peer P . It selects m blocks in this piece and produces one coded block x where:

$$x = \sum_{i=1}^m C_i^P \cdot b_i^P \quad (2.2)$$

The coding coefficients are selected to encode data blocks into x and be embedded in the header of the coded block. Thus, an overhead is imposed per coded block. Therefore, a random seed is responsible to generate a series of random coefficients by a pseudo-random number generator which efficiently decreases this overhead.

In downstream peers, Gauss-Jordan elimination method is implemented for the decoding process. In this method, the decoding process begins when sufficient number of coded blocks are received. The fundamental characteristic of randomized push mechanism with random linear coding is each data piece can be served by multiple seeds that are collaborated with each other without any protocol signaling. Therefore, each coded block is independent as any other block, regardless of the seed that generates them.

For the implementation, a cluster of 48 dedicated dual-CPU servers was built and servers are interconnected by Gigabit Ethernet links to evaluate the performance of R^2 protocol. Each peer in R^2 implementation operates two main processes: a network process to maintain all input and output UDP flows or TCP connections and channel properties, and engine process which is responsible of buffering incoming data blocks, sending coded blocks to out-bound connections and applying random network coding by generating random codes over $GF(2^8)$ finite field.

For the evaluation, a traditional pull-based protocol referred to as Vanilla is used with network coding for the sake of comparison against the R^2 protocol. The evaluation considers the following metrics: 1) Playback skips. 2) Bandwidth redundancy to measure the discarded data pieces blocks due to lateness or dependency over all received pieces or blocks in the playback buffer. 3) Buffering levels on each peer during a live streaming session. 4) The upload bandwidth utilization on the stream server.

R^2 has shown a constant playback quality with less than 0.02% of playback skips, where the percentage is higher in Vanilla with network coding when the number of peers increases.

The evaluation shows that 15% of the upload capacity of the streaming server is saved by R^2 . The buffering level ramped up sharply and remained stable in R^2 while Vanilla maintained a lower level with a bit variation over time, in terms of bandwidth supply and demand.

The R^2 protocol has performed better than Vanilla protocol when the supply match the demand, and also when the demand exceeds the supply. The R^2 protocol has been able to

conserve a steady buffering level around 90%, while Vanilla with network coding has struggled to keep the buffering level above the priority region.

In R^2 , network coding is applied on top of the application layer regardless of the underlying network infrastructure. In the previous evaluation, the authors have not mentioned the average computation process time in coding/decoding packets in peers which is an important parameter to characterize the efficiency of the used coding technique. In our work, we will focus on applying network coding across the network layer and migrate the complexity of buffering and peers synchronization to the centralized network controller entity.

In [20], Kim, Choi, and Park propose a heuristic and distributed mechanism called Centrality-based Network Coding Node Selection (CNCNS) to reduce the number of network coding nodes for any network topology in order to minimize the overall network coding overhead. They refer to a selected node as a central network coding node in an area or a set of nodes.

To compute the centrality property for a node in a network, two main parameters have been considered: (1) the degree which represents the number of flows that pass through a node from a source to receivers, and (2) the strength which is defined as the sum of links bandwidth that connect a node with its neighbor nodes. CNCNS algorithm is controlled by two weight parameters: α and β . The parameter α is used to compromise between the node degree and the node strength while β is used to tradeoff between packet transmission rate and packet innovativeness.

For the performance evaluation, the study used the following assumptions: single source node is connected to multiple sinks in a random network topology that consists of 50 nodes.

In addition, each sink node is connected to the source node via six intermediate nodes as a diameter. The arrival of packets follows the Poisson process. The network throughput is measured by the number of decoded packets. In the results, the throughput has increased proportionally when the number of network coding nodes are increased in the network.

For β value between 0.5 and 1, CNCNS selects the network coding node with more innovative incoming packets, and that increases the network throughput. In term of average end-to-end decoding delay, network coding causes an overhead in the decoding process because the receiver waits for sufficient number of independent or innovative packets to decode the original data sent. So, whenever more independent packets are received, the decoding delay is reduced. In this work, the authors do not consider the selection criteria of the area size or whether CNCNS mechanism can be extended in dynamic or heterogeneous multicast network topologies.

Xuan and Lea [19] introduce network coding as a solution for low throughput problem in non-blocking multicast networks. In a conventional non-blocking multicast network, edge nodes of the network perform the admission control of data traffic, without any coordination with the intermediate nodes to prevent any congestion inside the network.

Finding the optimum routing with minimum bandwidth consumption for multicast session in a non-blocking network is not an easy task, because there are many network parameters that determine the route feasibility. They have observed the most significant benefit cited in the study that the network coding treats a single multicast connection to t destinations as t unicast connections. As a result, they have proved the following points: 1) the optimal paths for source-destination pair in a non-blocking unicast network are also the optimal for the same pair in multicast non-blocking network. 2) The non-blocking multicast network

can admit the same amount of data traffic as the non-blocking unicast network. So, based on that, they have discussed analytically the optimal routing formulation of non-blocking multicast network with network coding for both explicit routing, and shortest-path routing. So, the problem of finding optimal routing is formulated as a linear optimization problem to minimize the bandwidth cost that is expressed as link congestion ratio r , where congestion ratio is a ratio between the amount of traffic routed through a link over the link's capacity. Also, this optimal routing should maximize the amount of admissible traffic θ that satisfies the ingress and egress constraints of the non-blocking multicast network. The study simulation compares the throughput between the legacy non-blocking routing and Constraint-based-Routing (CBR) approaches using a simulation environment with 15 nodes, 62 directed links and fixed bandwidth capacity value for each directed link. The results show the throughput of non-blocking multicast network without using the network coding for two routing algorithms: the shortest-path tree and the proposed optimal routing. The optimal routing achieves 20% higher throughput than the tree-based routing algorithm. In addition, they compare the throughput for two CBR schemes, the shortest-path tree SPT-CBR and network coding-based NC-CBR. The throughput of NC-CBR approach achieves 33% higher than SPT-CBR when the average number of receivers is 3 and around 30% when the average of number of receivers is 4. So, the study shows the significant benefits of network coding in increasing throughput for hard QoS grantees multicast networks.

In [16], Kontai *et al.* design and implement an SDN controller that supports IP multicasting and switching between multiple multicast trees with minimum packet loss, and without duplicate packets. The OpenFlow protocol is used to compute and assign multicast-trees in a centralized and programmable fashion, as shown in Figure 2.1.

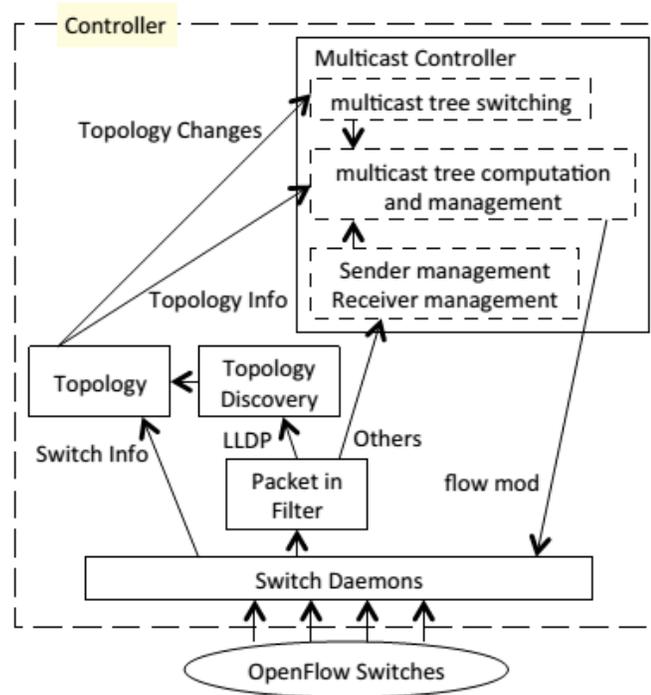


Figure 2.1 Overview of OpenFlow Controller proposal for IP multicast networking [19]

The main function of the proposed controller is to create flow entries in the forwarders e.g. switches to initialize multicast-trees in the network topology. The design has employed a number of modules to compute and manage these multicast-trees. These modules are summarized in the following:

- 1) Sender management module: to identify the senders' locations.
- 2) Receiver management module: to observe Internet Group Management Protocol (IGMP) packets from hosts and also store the receivers' locations.
- 3) Multicast-tree computation and management module: computes and discovers multicast-trees using the feedback from the previous modules and caches these trees information to reduce the routing initial time. Two multicast trees are used: one is the active tree that serves the delivery of packets, and the other is the backup tree that is used when a failure occurs in the active tree.

- 4) Topology daemon: obtains the network topology information and sends it to the other management modules.
- 5) Multicast tree switching module: receives the state messages from all network switches, determines the multicast groups that have been affected by a failure, and switches multicast groups to unaffected trees utilizing the obtained information from this module.

In the implementation phase, the designed OpenFlow controller creates and caches two multicast trees created using Dijkstra's algorithm. It assigns a unique ID for each tree where this ID identifies each tree in multicast group and it is embedded in the header of packet when the packet traverses the multicast network. The controller creates flow entries for active and redundant trees inside the multicast switches, except the switches that the senders are connected to. When a host joins or leaves a multicast group, the controller modifies the flow table entries of all trees that belong to the same multicast group.

The authors have evaluated their proposed controller to verify it can switch faster between trees with minimum packet loss for three different network scenarios composed of 5, 7, and 9 OpenFlow compliant switches, respectively, with one sender and two receivers for all three scenarios. A 30 Mbps video stream has been sent by a sender to a multicast group, the stream is Real-time Transport Protocol (RTP) packets. They occasionally cause a failure in the network to let the controller switch the data traffic between the trees by shutting down the port that is close to the root of the tree. The handover time and packets loss are measured and the values are counted by monitoring the sequence number in RTP headers and the time between the last received packet before the failure has been introduced and the first packet received after the end of the handover process.

The results show the minimum tree switching time is 13.3 msec and the maximum packet loss is 1 packet for all three network scenarios. The video stream frame rate is 30 frames/sec, so each frame is sent every 33.3msec into the network so the tree handover time is shorter than the frame rate. The delay between the OpenFlow controller and switch response is 0.63msec on average for 30 trials.

We will use the design of [16] to develop a compliant RLNC aware controller for handling network coding in real-time IP multicast networks.

Liu and Hua [17], propose a framework for realizing network coding (NC) over SDN across the network layer as illustrated in Figure 2.2. In this framework, there are four main functions that include both the SDN controller and the switch: 1) Initialization function that appends NC header to each encoded packet, 2) Encoding function for buffering and encoding packets, 3) Decoding function to compute the original packets by solving the system of linear equations. 4) Output function in the switch for forwarding packets. The controller is responsible to compute the multipath multicast tree, selecting encoding scheme, and generating NC flow entries to be pushed into the respective switch.

In multipath multicast routing, max-flow algorithm is applied to produce a subgraph that contains disjoint paths of minimum cost links between each sender-receiver pair.

In the produced subgraph the end-to-end hosts are excluded and they are not even under the control of the SDN controller. For encoding scheme, some deterministic and random encoding algorithms are proposed to find the local encoding matrix or encoding coefficients that satisfy the linear independency of global encoding vectors of all received packets at each receiver node. The generation function of NC flow entries executes two steps: 1) Mapping the assigned function to an action or a list of actions, 2) Generating NC

flow entries with parameters that are needed to execute the corresponding actions. There are two types of buffers are proposed to perform network coding: packets buffer and status buffer. Packets buffer is used to store the received packets that are required to be encoded or decoded. Status buffer is used to record generation ID and count the cached packets of each generation. The generation is the group of packets that are combined or encoded together by the same encoding code. Both packet and status buffers are managed by the controller to alleviate the buffer management overhead from the switch.

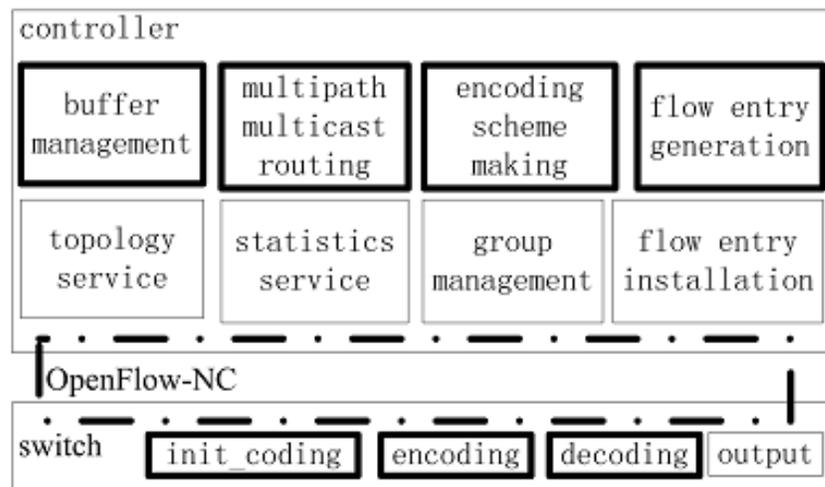


Figure 2.2 NCoS framework

In the implementation phase, the authors have extended OpenVSwitch to include coding and decoding actions, and extending POX SDN controller to include multipath multicast routing, encoding schemes, NC flow generation, and buffer management modules. The framework is tested on the Mininet network emulation platform. The results confirm the network coding computational complexity is very high, especially, when random coding algorithm is implemented. The authors have selected XORing as a simple and fast coding operation that could reduce this computational overhead. The XORing encoding scheme is applied on 20 to 80 nodes using TopGen emulation lab to generate random topologies. The

outcomes show the computational overhead decreases by more than 17.5% and XORing method has used 50% of CPU utilization relative to the randomized coding method.

Szabo *et al.* [18] analyze and measure the latency and the frequency of packet retransmission for three network coding schemes (End-to-End, hop-by-hop and RLNC) by implementing these schemes in a software router.

They have discussed analytically the improvement in latency and packets retransmission in the mentioned network coding schemes for constrained linear multi-hop network topology. In addition, they have built a full-fledged implementation environment to measure the overall exchanged packets, packets loss, and average delay between a source and destination.

The authors assumed each packet suffers the same propagation delay on each link, and they ignored the consumed time in buffering, coding, and decoding packets.

In the implementation, they built a software router that uses ClickOS as a modular router platform to virtualize the network functions such as packet classification and scheduling. They have implemented Kodo library to develop a RLNC encoder, recorder and decoder.

The SDN controller is not used in their implementation, and a software router is made to perform all networking and encoding/decoding functions.

The measurements show theory results match simulation results. Hop-by-hop (HbH) and RLNC have the same number of retransmission packets and it is less than that for the end-to-end (E2E) scheme where the number of retransmission packets increase linearly with the packets loss.

The experimental prototype is designed to integrate NC and SDN with ESCAPE [24] platform which is capable to perform OpenFlow functions to implement NC schemes using Virtualized Network Functions (VNF).

Finally, we summarize the previous frameworks for the integration between SDN and NC and illustrate the gaps as shown in the following table.

Framework/prototype	NC schemes	Gaps
NCoS [17]	- XORing , RLNC	<ul style="list-style-type: none"> - Computation overhead is ignored as a measurement parameter. - No analytical references to compare the results with. - The implementation was limited to XOR encoding function.
NC-SDN using ESCAPE [18]	- RLNC	<ul style="list-style-type: none"> - Evaluation was only for linear multi-hop topologies. - Computation overhead is not assessed. - No SDN controller is used in their prototype.

Table 2.1 Summary of the literature NC-SDN frameworks

CHAPTER 3

NC-SDN EXTENDED FRAMEWORK

3.1 Overview

In this chapter, we developed an extended framework of network coding integration with Software Defined Network architecture. Figure 3.1 illustrates the proposed integration framework.

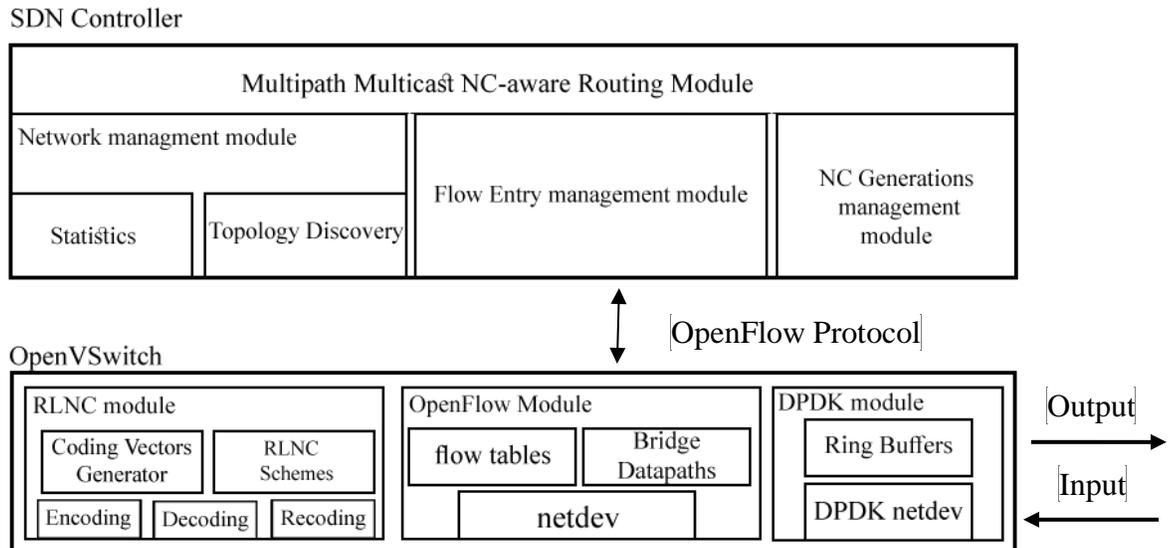


Figure 3.1 The proposed SDN-NC framework architecture

We classified the extended components based on the two main SDN architecture planes:

3.2 Data plane NC integration

Data plane is the network structure that responsible of forwarding, processing and performing the control plane actions on the network data flows. Therefore, the network coding functions are required to be applied into this plane to perform encoding, decoding

and recoding operations on the multicast packets that traverse the network via network devices such as switches or routers.

The switch (hardware or software) is the active network element that implements the data plane structure to process and forward a packet from one node to another. For this work a software SDN compliant switch is extended to include the network coding capabilities.

In this section, we discussed in detail the extended and added modules:

3.2.1 OpenFlow Module

OpenFlow is the de facto protocol of the communication interface between the two parts of SDN paradigm. It allows the controller to manage, configure and interact with data plane components.

The OpenFlow protocol consists of four components:

Message layer

Message layer defines the protocol core messaging structure and the semantics of all messages, and it supports messages construction and manipulation.

The OpenFlow message consists of a header and payload. The header structure is unified in all OpenFlow messages and it has four fields: version, type, length and a transaction identifier (xid).

The version field identifies the OpenFlow protocol version where the message belongs to, type field defines the message type, and the length field indicates the length of message stream bytes, finally, the transaction identifier is a unique value used to match the requests and responses that are exchanged between the controller and the switch. The Figure 3.2 below illustrated the OpenFlow message header format.

The OpenFlow standard provides the flexibility for vendors or researchers to extend the protocol message layer to support their customized OpenFlow messages. Therefore, the vendor or experimenter message type is introduced in OpenFlow protocol for this purpose, so we utilized this message type to extend OpenFlow protocol to support network coding functions. The Figure 3.3 shows the structure of experimenter message of OpenFlow version 1.3.

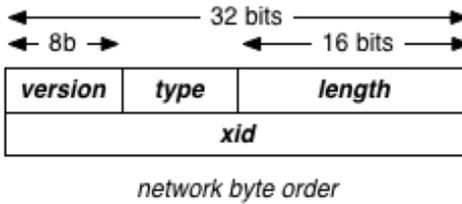


Figure 3.2 OpenFlow header structure

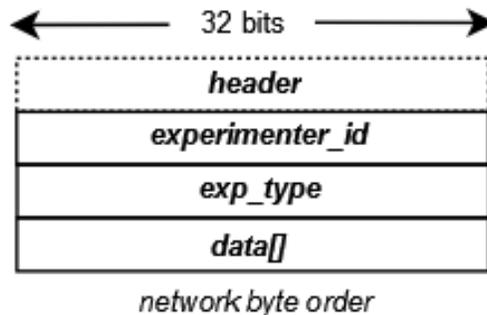


Figure 3.3 OpenFlow experimenter message structure

State Machine

OpenFlow protocol has a simple finite state machine to maintain the message exchange between the controller and the forwarding entities and it handles all messages asynchronously.

the state machine is illustrated in Figures 3.4 and 3.5 for both the controller and switch perspectives. However, the protocol connection establishment includes the version and capabilities negotiation between the controller and the switch. After the version negotiation is successful between both ends, the controller starts the features discovery phase. So,

through the features discovery phase the controller will be aware of network coding enabled switches to manage and configure network coding related modules.

The sequence diagram of OpenFlow connection establishment and features discovery phase are illustrated as in Figure 3.6.

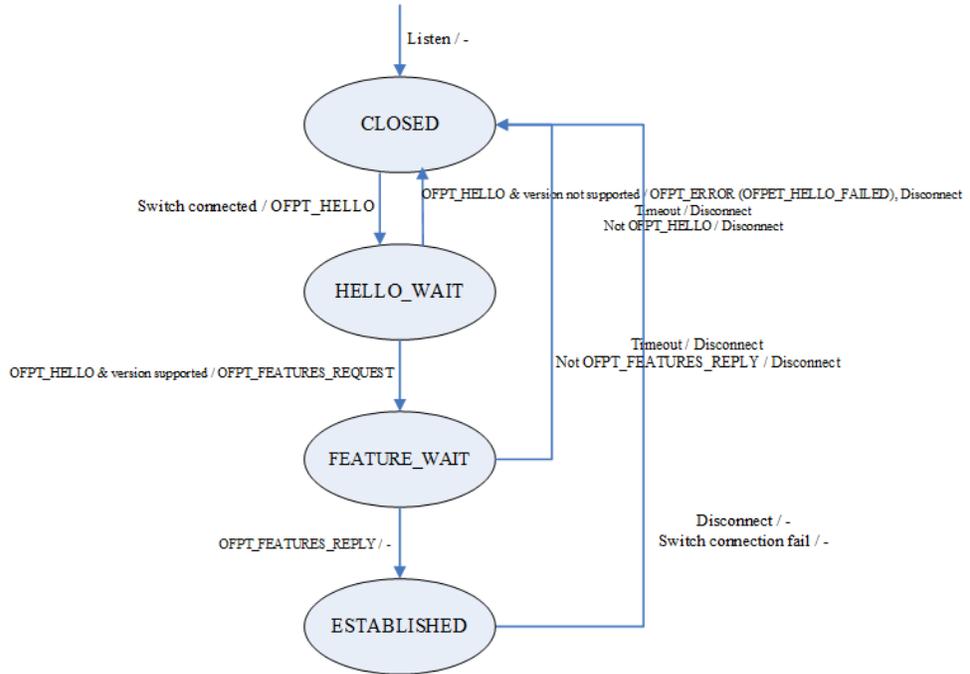


Figure 3.4 Controller connection state machine

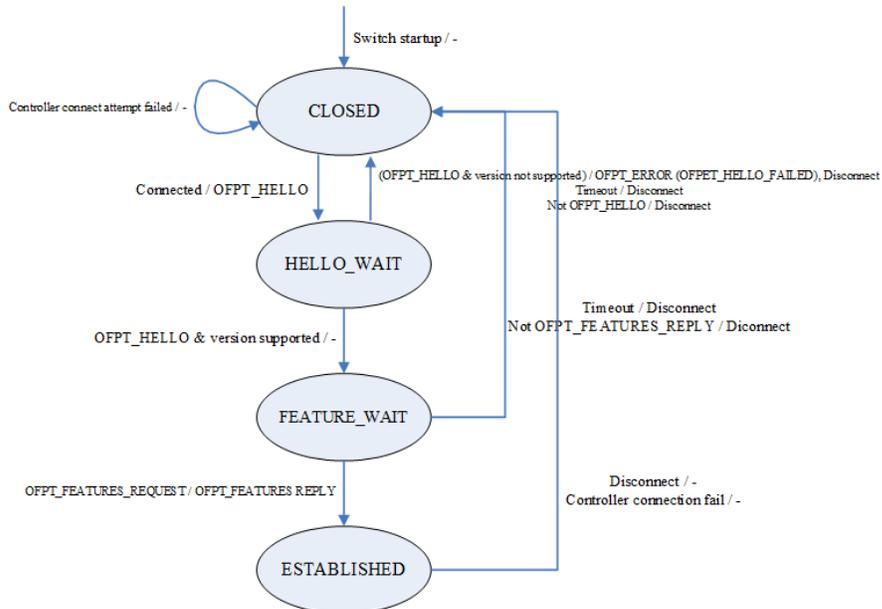


Figure 3.5 switch connection state machine

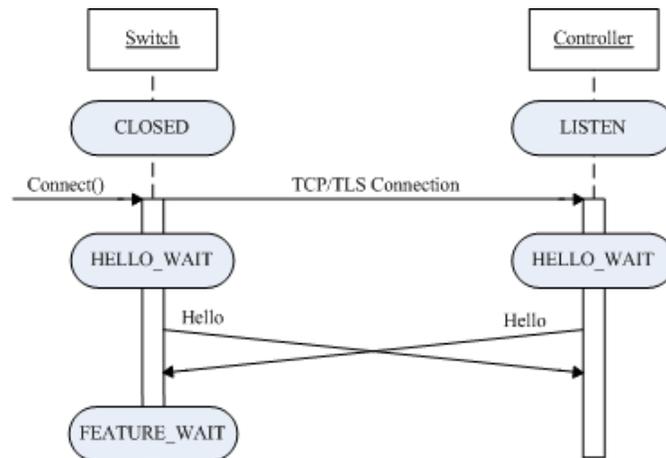


Figure 3.6 OpenFlow connection establishment phase

System Interface

System interface provides the service interface between OpenFlow protocol components. OpenFlow system interface consists of four main interfaces: TCP/TLS interface, switch agent interface, controller application interface and the configuration interface.

TCP/TLS interface provides an oriented connection between the controller and the switch.

Switch agent interface interacts with the switch kernel system and exchanges messages asynchronously with the controller. Controller application interface interacts with the higher-level controller applications that run on top of OpenFlow protocol stack, and it exchanges messages between them. The configuration interface allows the network operator to configure OpenFlow protocol parameters.

Configuration

Configuration component provides the language and utility for configuring the controller and switches and validating the syntax using a front-end compiler.

3.2.2 OpenFlow Network Coding Messages

Network coding feature discovery

After the TCP/TLS connection between the controller and the switch is established, the controller sends FeatureReq OpenFlow message to the switch to recognize the capabilities and actions that can be performed by the switch. The featureReq message has only a header with FeatureReq value type. Next, the switch replies to the controller with FeatureRes message which contains the switch datapath ID, number of PacketIn buffers, number of flow tables, switch supported features and actions, as shown in Figures 3.7 and 3.8.

In our case the network coding enabled switch would response with one or more network coding actions: encode, recode or decode.

From OpenFlow 1.1 and followed versions, FeatureRes message does not advertise the supported actions directly as in previous OpenFlow 1.0.

In our case, we have defined network coding functions as optional actions. The optional actions can be retrieved from a flow table known as Stats table using StatsRes message.

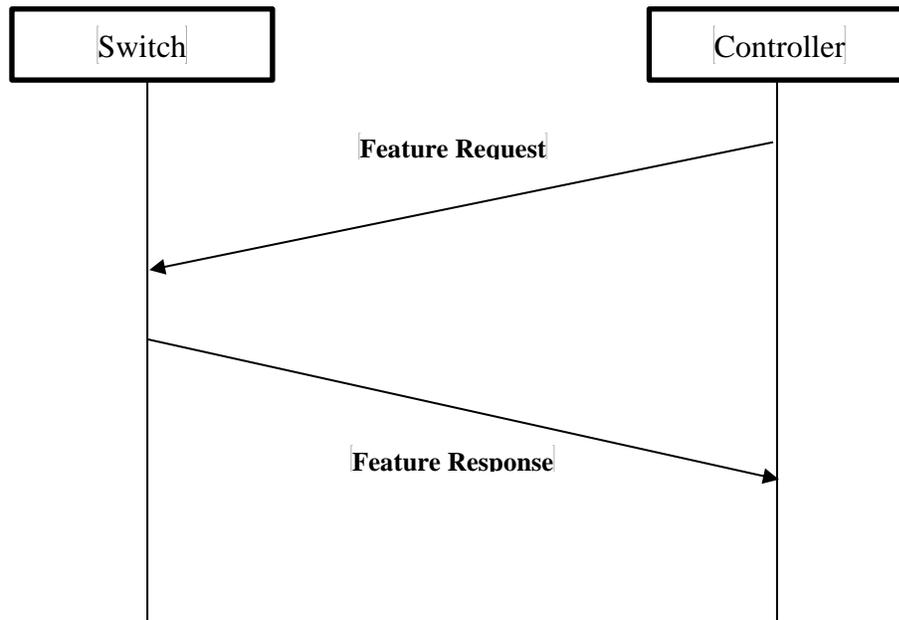


Figure 3.7 OpenFlow features discovery phase

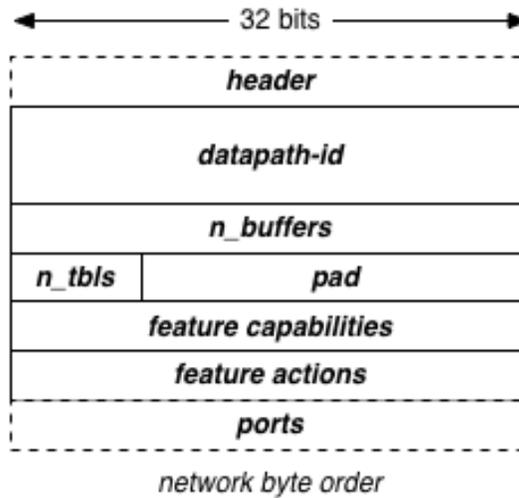


Figure 3.8 OpenFlow FeatureRes message structure

3.2.3 Network coding Flow Modification Messages

- **Encode/recode message format**

The controller can send proactively or reactively encode action message to any supported OpenFlow switch. Encode action message pushes a setup entry in the switch flow table to apply RLNC action on the matched packets and forward them to the designated output ports.

Figure 3.9 shows the structure of encode action message and the required parameters to perform encoding function.

In following, we will describe each message fields in details:

- **Network Coding Feature Identifier:** A unique ID to identify the message type as an experimental OpenFlow message and recognize NC instructions group.
- **Encode/Recode Action Identifier:** A unique value to differentiate between NC instructions (encode, decode and recode).

- **RLNC schema type:** this field is used to configure how coding vectors are generated and what kind of finite field is chosen to perform network coding actions.
- **Input port:** the port where packets are received from.
- **Output port:** the port where the encoded packets will be distributed.
- **Number of Buffers:** the required number of buffers that hold packets for coding process.
- **Max symbol size:** defines the maximum symbol size to be buffered and processed.

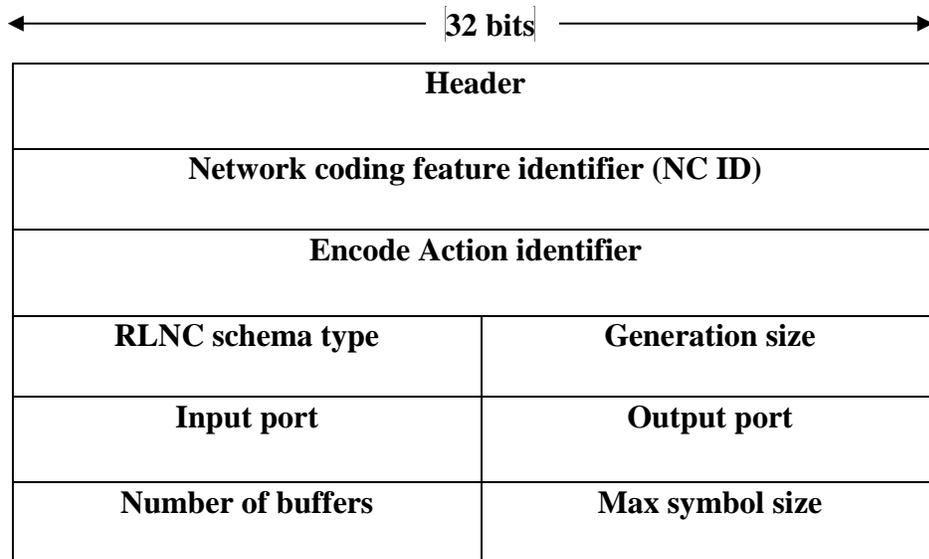


Figure 3.9 OpenFlow Encode action message structure

- **Decode message format**

To enable decoding process at a receiver node, the controller composes a decode action message to impose the decoding process in that node.

The following Figure 3.10 illustrates the message structure of decoding action.

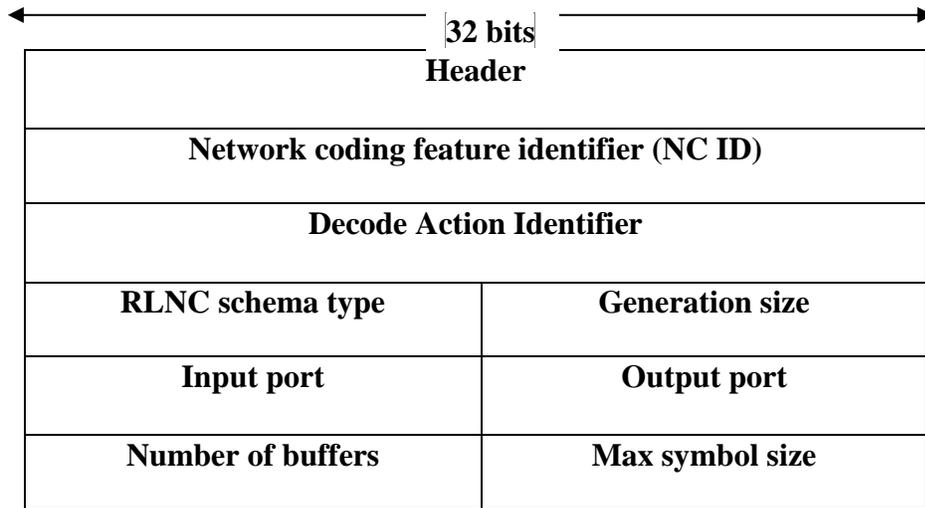


Figure 3.10 OpenFlow Decode action message structure

3.2.4 OpenFlow Matching

The flow matching process is the cornerstone feature in SDN OpenFlow compliant switches.

The OpenFlow switch has flow tables that contain entries to match packets with certain conditions or values, and perform the corresponded actions such as regenerating headers, identifying next-hop destination, encapsulating packets or any other type of packet processing in a pipeline behavior. OpenFlow pipeline emulates the hardware pipelining where one table can be used to perform port lookup for example and another table to manipulate the packet header at the same time. However, multiple OpenFlow tables are composed to perform multiple tasks on the matched packets before forwarding them out.

Basically, to support network coding processes we must define custom flow entries with the required matching fields to recognize the coded packets and the multicast packets that require encoding. In addition, manipulate packets headers to match the designated routes to their destination.

Therefore, custom OpenFlow matching fields are created to identify NC packets and the parameters that are involved in encoding, decoding and recoding actions. We have utilized the proposed custom fields in NCoS framework [20] as shown in Table 3.1.

Initially, we have simplified and modified these flow rules, and decreased the number of required flow matching fields by migrating the buffer management from the controller side to the switch itself, and handling them using DPDK module, where we will explain it later in the buffer management section. Our modified NC flow rules are depicted in Table 3.2.

Field	Description	Related actions		
Buffer id	Buffer for storing packets	Config	Encode	Decode
Status buffer id	Buffer used for storing shared status	Config	Encode	Decode
Output num	Number of output ports	Config		Decode
Generation size	Number of packets in a generation	Config		
Input num	Number of input ports		Encode	Decode
Output port	Output port index		Encode	
Output list	List of output ports	Config		Decode
Code vector	Local code matrix	Config	Encode	

Table 3.1 NCoS framework OpenFlow matching entries

Field	Description	Related actions		
Generation size	Number of packets in a generation	Config		
Generation Id	The generation identifier the packet belongs to.		Encode	Decode
Input port	Input port index		Encode	Decode
Output port	Output port index		Encode	
Coding scheme	RLNC coding scheme	Config		
Max symbol size	Maximum symbol size	Config	Encode	Decode
Buffers num	The required number of buffers to store the packets	Config		

Table 3.2 NC-SDN implementation OpenFlow matching entries

3.2.5 OpenFlow Extensible Matching

A flexible matching structure known as OpenFlow Extensible Matching (OXM) is developed to replace the old rigid OpenFlow matching mechanism in versions 1.0 and 1.1 as depicted in Figure 3.11.

OXM is introduced in OpenFlow 1.2 in a simple TLV structure (Type -Length -Value) where the matching entries are identified by class or type, the length that indicates the size of the payload in bytes, and succeeded by the value of the payload. The 4-bytes OXM header has a single bit to identify the bitmask existence in the payload as shown in Figure 3.12. OXM entries with pre-requisites should be enrolled after the requisites OXM entries. For example, the IPv4 ToS field must be preceded by EtherType = 0x8000 OXM entry.

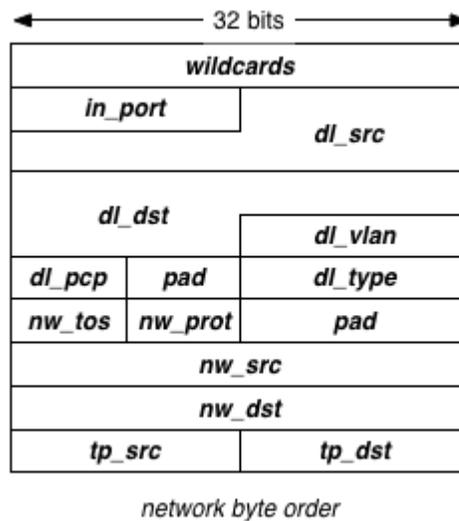


Figure 3.11 OpenFlow 1.0 fixed matching structure

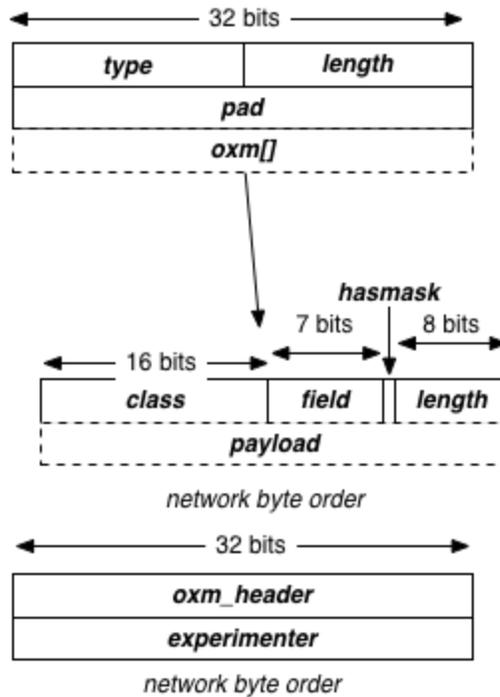


Figure 3.12 OpenFlow Extensible Matching structure

In our implementation, four matching entries are included and listed as following:

- **OFPXMT_OFB_ETH_TYPE:** is used to detect the type of Ethernet packet. Ethernet type is a significant field in our design because it indicates the type of flow packets whether if they are coded packet or original (non-coded) packets.
- **OFPXMT_OFB_IPV4_SRC:** is used to detect the source IP address of the packet that originated from a multicast source.
- **OFPXMT_OFB_IPV4_DST:** is used to match the multicast destination IP address of the packet that generated from a multicast source.
- **OFPXMT_OFB_NC_HDR:** this matching value detects the required network coding fields, where mentioned before in network coding OpenFlow modification messages section. The required matching network coding header fields are following:

- **Generation ID:** is a unique ID to identify the group of coded packets and forward them to the destination for decoding.
- **Input port:** the identifier of the ingress port of the switch.
- **Output port:** the identifier of the egress port to the next hop.

The prerequisite of this OXM entry is OFPXMT_OFB_ETH_TYPE which is responsible of detecting the coded packets that are encapsulated into Ethernet frames.

3.2.6 OpenFlow Instructions

OpenFlow instructions are triggered after the flow matching process. There are six different types of instructions in OpenFlow 1.3 as described below:

- **Apply Actions**

This instruction is utilized to perform immediate actions on the matched packets. We use this instruction to perform encoding, decoding or simple forwarding actions.

- **Write Actions**

This instruction lists actions to be performed later.

- **Clear Actions**

This instruction is used to clear the accumulated actions list.

- **Meter**

Meter instruction is only for updating flow meters that are used to provide statistical information to support high-level network control applications.

- **Goto Table**

Goto Table instruction is used to send a packet to another table in the switch to match different OpenFlow rules.

- **Write Metadata**

Some data can be stored or attached to the packet as it traverses the flow tables to help in packet matching process from one table to another.

The structure of instruction messages is a TLV type as shown in the following Figures 3.13 and 3.14.

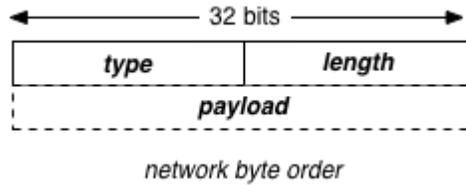


Figure 3.13 OpenFlow TLV structure

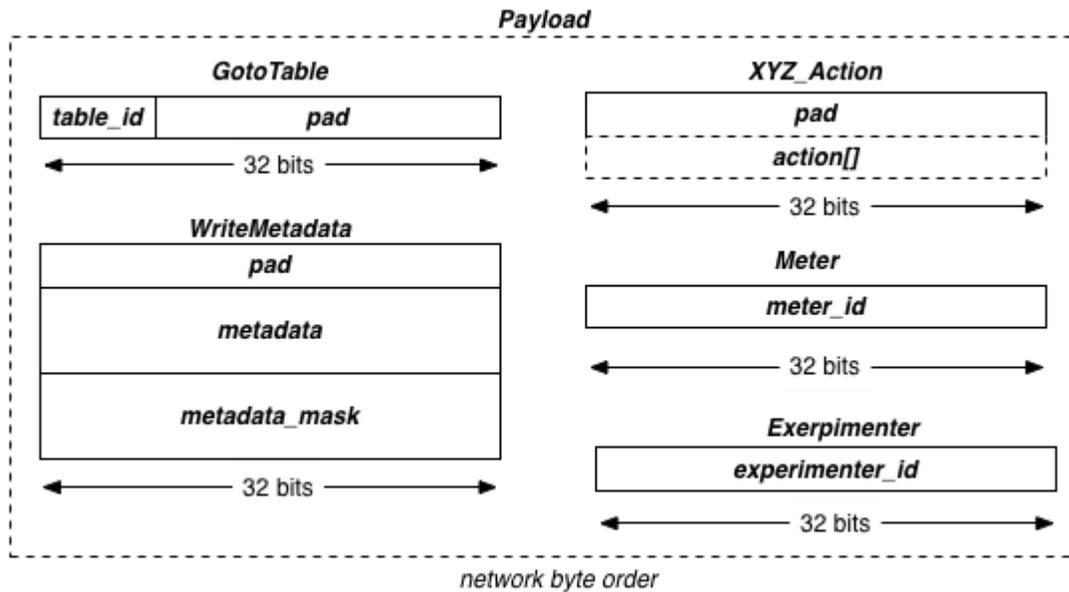


Figure 3.14 OpenFlow instructions in a TLV payload

In our network coding OpenFlow datapath extension, we introduced three new actions, defined as following:

- **Configuration action**

This action configures the NC enabled switch with the required parameters that are mentioned early in Flow Modification Messages section.

- **Encoding action**

This action executes network encoding on a group of original packets (generation) that belong to a single multicast session, then encapsulates the resulted new coded packet into an Ethernet frame with Ethernet type field value 0x8877 as a unique NC frame identifier.

- **Decoding action**

In this action, the coded packet is extracted from the Ethernet frame and buffered to be decoded with the other coded packets of the same generation.

Once the sufficient number of independent coded packets are received, the switch regenerates all original packets and update the controller to mark this generation as received.

3.2.7 Network coded packet structure

NC packet header is constructed after the coding process is performed on the generation's packets. The structure of the network coding packet is illustrated in Figure 3.15. The header is like the proposed header in NCoS framework [20].

The encoded packet is encapsulated in Ethernet packet with ether type value 0x8877 as shown in Figure 3.16.

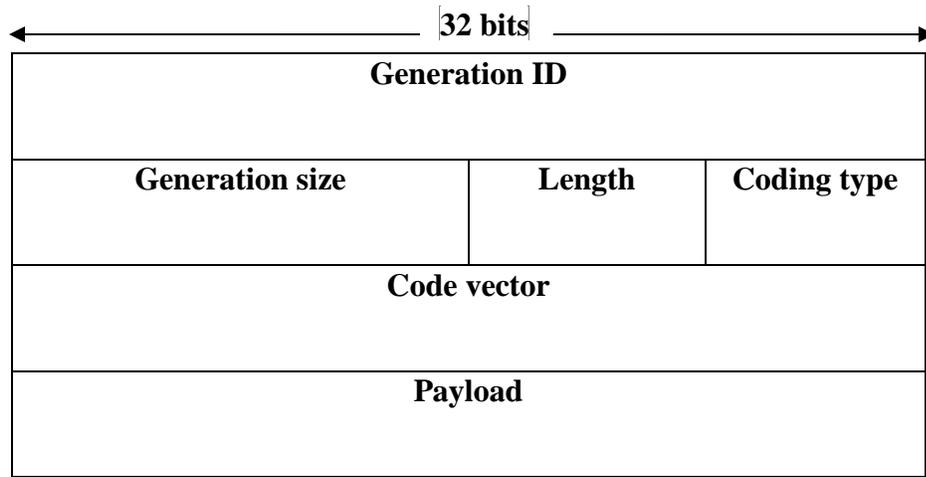


Figure 3.15 Proposed network coding packet structure

Preamble	SFD	Destination MAC address	Source MAC Address	Ether type 0x8877	NC packet	FCS
-----------------	------------	--------------------------------	---------------------------	--------------------------	------------------	------------

Figure 3.16 Ethernet frame encapsulation of NC packet

3.2.8 DPDK buffers management module

The generic design of OpenFlow datapath architecture resulted performance limitations in specific use cases in packet switching and packets processing [20] [21]. Network coding implementation is one of these cases where encoding/decoding process is a latency bottleneck as mentioned before in the literature. Packets buffering is essential in packets handling and it requires efficient and optimized mechanisms to avoid high-latency between switch ports and packet processing modules e.g. Memory zero-copy and pipeline processing.

Data Plane Development Kit (DPDK) [25] is a software framework that has been adopted in our software defined network coding implementation as a buffer management component and network coding accelerator. DPDK is a set of drivers and software libraries that enables fast packet processing for general-purpose multi-core hardware and supports many NICs. DPDK provides important features such as lockless queues and memory allocation management. The general architecture of DPDK is illustrated in Figure 3.17.

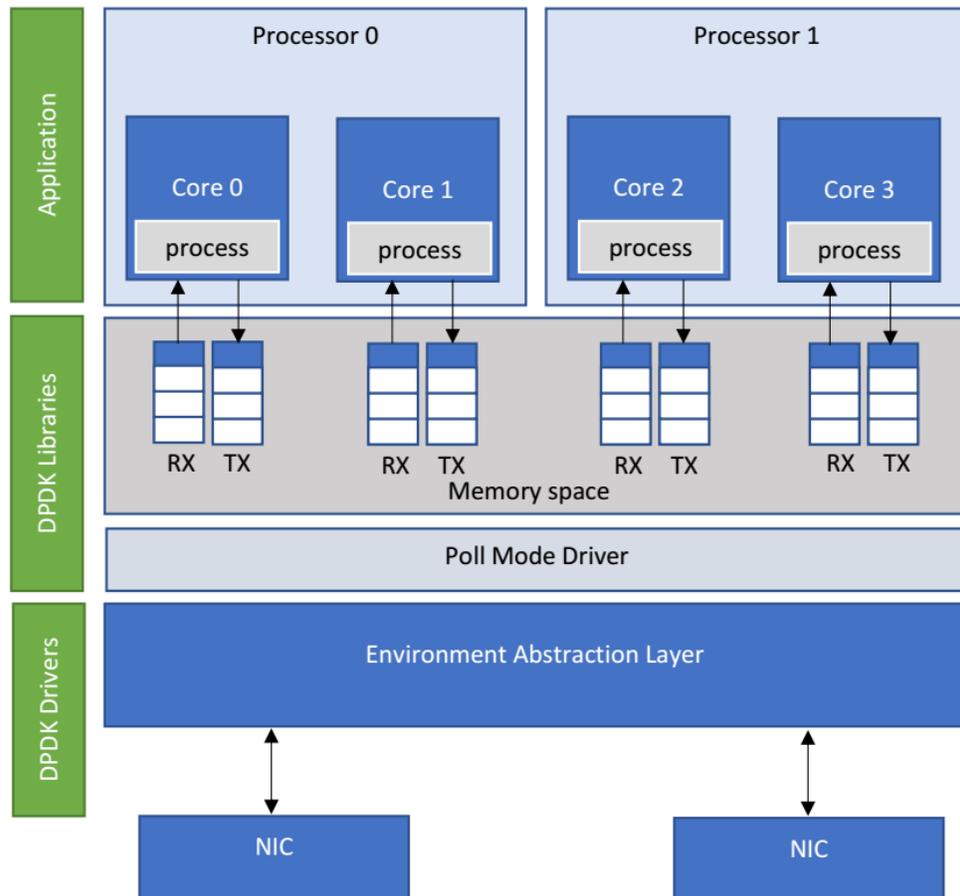


Figure 3.17 Data Plane Development Kit Architecture

3.2.8.1 Environment Abstraction Layer (EAL)

DPDK framework creates a generic interface that gives the user-level application direct access to low-level resources such as physical network ports and memory space. EAL hides the environment specifics from libraries and applications. The EAL provides services such as system memory reservation, core assignment, PCI address abstraction and interrupt handling.

3.2.8.2 Poll Mode Driver (PMD)

Poll Mode Driver includes APIs to retrieve packets from network port buffers to processing CPU cores through packet queues. PMDs are designed to work with per-core private

resources. For example, a PMD maintains a separate transmit and receive queues per-core, per-port to avoid lock contention.

PMD accesses the RX and TX buffer descriptors directly without any interruption to quickly receive, process and deliver packets in the user's application.

3.2.8.3 Memory Ring Buffers

The ring buffer is a circular data structure that enables lockless bulk or burst queue/dequeue packet operations. In our implementation, we utilize ring buffers to queue multicast packets of specific burst or generation for encoding and decoding operations.

3.2.8.4 DPDK network coding functionality

When a multicast packet reaches a switch port buffer, the packet header data is retrieved and compared with OpenFlow table entries, if the header matched an entry it will be buffered in a queue for encoding/decoding or just to be forwarded. The queue is a fixed size ring buffer, and it is allocated and assigned by EAL to a specific logical core that responsible to consume or process packets in the queue. The logical core executes the corresponding flow entry action on a packet or a group of packets. When a packet is generated or processed the logical core pushes it into a transmitter queue where it is assigned to an output switch physical port buffer as depicted in Figure 3.18.

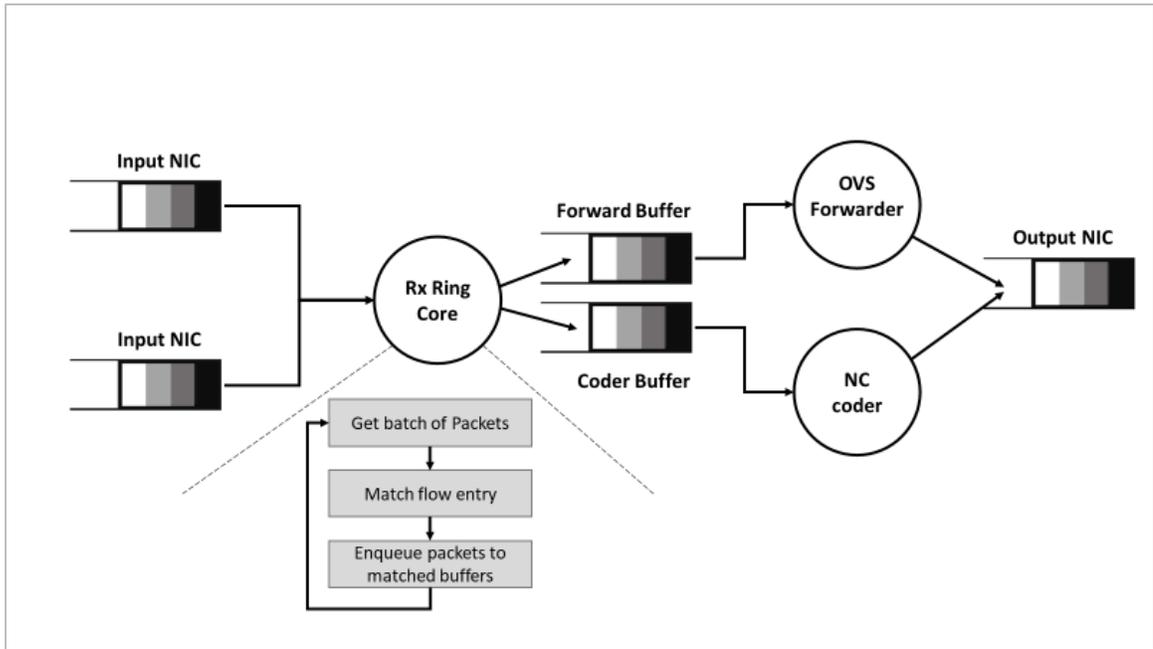


Figure 3.18 NC packets buffering in DPK module

In a receiver (decoder) node, DPK handles the received encoded packets and classifies them per generation ID. each generation has a buffer assigned to a logical core that computes the original packets by solving equation systems using the received NC packets and their headers code vectors. Then, the core forwards the recovered packets to the matched destinations as shown in Figure 3.19.

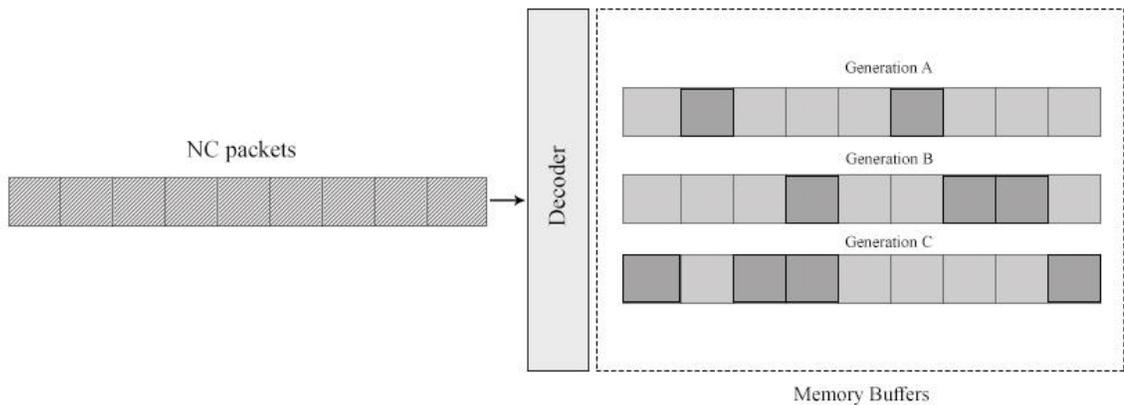


Figure 3.19 DPK buffers in a NC decoder node

3.2.9 Random Linear Network Coding Implementation

The potential of RLNC as a distributed network coding scheme that it can be practically implemented in multicast or broadcast applications to improve the network performance in terms of throughput, reliability and energy efficiency.

In wireless broadcast or multicast applications, RLNC performs better especially in harsh environments and it shows more robustness, reliability and a few number of retransmissions. In wired networks such as LANs or SONETs, the wired links are more reliable than in the lossy wireless networks and they provide a robust collision-free transmission. On the other hand, there are many overheads exist in wired networks like multiple access, multipath routing and multicast group management. The flexible SDN architecture is capable to fit network coding functions in current multicast networking and coexist with the overheads and complexities.

In this section, we will explain in detail the implemented RLNC module into SDN forwarding entities. Before that, we must define some fundamental RLNC annotations as following:

Generation

Generation is a block of packets or symbols that are encoded by one or more coding coefficients that are chosen randomly and independently from a finite field to generate a single coded packet or symbol. However, a single generation can produce any number of encoded packets. The previous studies have shown that the generation size has a significant effect on the performance of network coding and decoding complexity.

Encoder

Encoder is the entity that performs coding operations on original packets or symbols for a given generation and transmits the encoded linear combinations through a channel.

Decoder

Decoder is the entity that recovers or reproduces the original packets/symbols after receiving a sufficient number of independent coded packets/symbols.

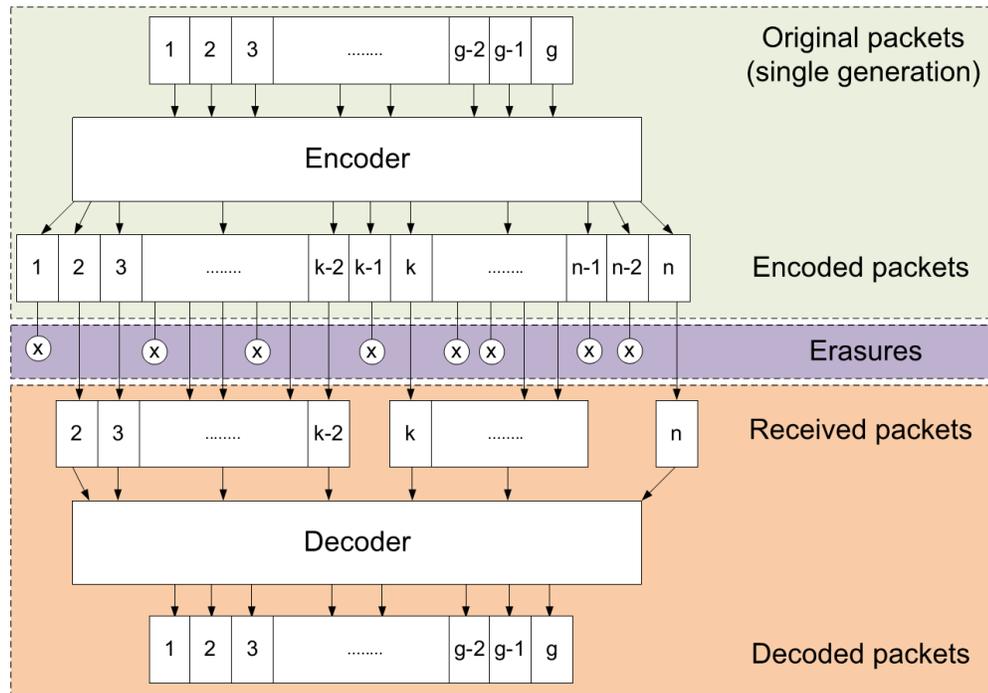


Figure 3.20 RLNC Encoding and Decoding process

Encoded packet

A packet or symbol that is generated after linear operations in a finite field are performed on some original packets or symbols that belong to a single generation.

Recoded packet

A recoded packet is a new linear combination of coded packets that are originally received at the intermediate network nodes. The new recoded packet can be generated from partially decoded generations. Recoding process can substantially increase number of innovative packets which received from the other nodes, and also improve the end-to-end decoding probability.

Matrix rank (R)

The encoding, recoding and decoding processes are performed via matrix operations. the rank of the matrix is the maximum number of independent rows (or the maximum number of independent columns) of a matrix. Calculating the rank is important to characterize the matrix and determine the number of decoding linear system equations.

Innovative packet

An independent packet that increases the rank or a decoding matrix.

3.2.9.1 RLNC operations

In this section, we will describe RLNC processes, encoding, recoding and decoding, and how they are performed in simple algebra examples.

As we have introduced previously, RLNC arithmetic operations are applied on Galois Finite Field $GF(2^m)$ elements where m defines the size of the field. In data communication systems, a symbol or a word is represented as a single byte (8-bits).so primarily when $n = 8$, $GF(2^8)$ is the base finite field that is used for all bitwise arithmetic operations where result elements or symbols in the same finite field.

3.2.9.2 RLNC Encoding process

The data source generates a packet B_i consists of a sequence of bits. The packet can be divided into S symbols or bytes. Where,

$$S = \text{packet length} / \text{symbol size.}$$

To code a new coded packet X_j , the encoder chooses randomly and independently a set of coefficients in $GF(2^8)$. For each original data packet, a one vector of coefficients known as coding vector is formed to compute a single coded packet as described below.

$$X_j = \sum_{i=1}^N C_{ji} \cdot B_i$$

Where C_{ji} is a coding coefficient and N is the total number of packets. This summation expression can be transformed into a matrix expression as following:

$$X = C \times B$$

Before sending the generated coded packet, the encoder attaches the coding coefficients vectors to the header of the coded packet which is represented as a vector called the information vector (C_j, X_j) .

The definitions of the arithmetic operations in Finite Field $GF(2^8)$ differ from the basic applied arithmetic operations such as addition, subtraction, multiplication and division. In the following we will discuss the required arithmetic operations in this Field to generate coded packets or symbols.

Addition in $GF(2^n)$

To add or subtract two field elements, you just apply a simple bitwise XOR operation. For example, to add 31 to 28 in $GF(2^8)$ it is equal to 3. also, we can express the addition operation in the polynomial form as following:

$$\begin{aligned} 31 &\rightarrow x^4 + x^3 + x^2 + x + 1 \\ 28 &\rightarrow x^4 + x^3 + x^2 \\ 31 + 28 &\rightarrow [(x^4 + x^3 + x^2 + x + 1) + (x^4 + x^3 + x^2)] \text{ mod } 2 \\ &\rightarrow [2x^4 + 2x^3 + 2x^2 + x + 1] \text{ mod } 2 \\ &\rightarrow x + 1 \\ &\rightarrow 3 \end{aligned}$$

Multiplication in $GF(2^n)$

The multiplication operation is more complex, but it can be implemented efficiently in the hardware or software. To multiply two field elements, the first step is to multiply their corresponding polynomials as in the basic algebra (except the addition is an XOR operation), and the coefficients of their polynomials terms are either 0 or 1 which makes the calculation easier. The result would be up to degree 14 polynomials which is larger than one-byte size element. So, In $GF(2^8)$ a fixed eight-degree irreducible polynomial (a

polynomial that cannot be factored into the product of two simple polynomials) is used as a divider of the intermediate polynomial product. The remainder of the division operation is the desired final polynomial element in GF (2⁸) Field.

Now let's try to compute the product of the same two elements 31 and 28.

$$\begin{aligned}
 31 &\rightarrow x^4 + x^3 + x^2 + x + 1 \\
 28 &\rightarrow x^4 + x^3 + x^2 \\
 31 \times 28 &\rightarrow (x^4 + x^3 + x^2 + x + 1) \times (x^4 + x^3 + x^2) \\
 &\rightarrow x^8 + x^6 + x^5 + x^4 + x^2 \text{ (intermediate product)} \\
 &\rightarrow (x^8 + x^6 + x^5 + x^4 + x^2) \% (x^8 + x^4 + x^3 + x + 1) \text{ (division over eight-degree irreducible polynomial)} \\
 &\rightarrow x^6 + x^5 + x^3 + x^2 + x + 1 \text{ (the Remainder)} \\
 &\rightarrow 111
 \end{aligned}$$

In the example, we divided the intermediate product over the irreducible polynomial $(x^8 + x^4 + x^3 + x + 1)$ that used in the Advanced Encryption Standard (AES).

Example of Encoding in RLNC

Suppose we have a generation of 4 data packets; each packet is size of 8 bytes as shown on the table below.

Packet 1	1	3	5	4	2	6	9	7
Packet 2	11	10	13	13	17	18	20	21
Packet 3	23	24	26	31	33	35	34	25
Packet 4	27	32	30	39	38	19	40	41

The randomly selected coding vectors are as following:

Coding vector 1	41	35	111	132
Coding vector 2	214	225	108	214
Coding vector 3	82	73	241	144
Coding vector 4	183	235	187	233

To generate the first coded packet, the coding vector 1 is used to encode each four column bytes of the four original packets to compute one coded byte of the coded packet as shown below.

$$\begin{aligned}
\text{1st encoded byte: } [41 \ 35 \ 111 \ 132] \times \begin{bmatrix} 1 \\ 11 \\ 23 \\ 27 \end{bmatrix} &= 41 * 1 + 35 * 11 + 111 * 23 + 132 * 27 = 176 \\
\text{2nd encoded byte: } [41 \ 35 \ 111 \ 132] \times \begin{bmatrix} 3 \\ 10 \\ 24 \\ 32 \end{bmatrix} &= 41 * 3 + 35 * 10 + 111 * 24 + 132 * 32 = 234 \\
\text{3rd encoded byte: } [41 \ 35 \ 111 \ 132] \times \begin{bmatrix} 5 \\ 13 \\ 26 \\ 30 \end{bmatrix} &= 41 * 5 + 35 * 13 + 111 * 26 + 132 * 30 = 225
\end{aligned}$$

This encoding process is repeated for all 8 column-bytes to generate the all required 8 bytes of the coded packet. However, the all arithmetic operations are applied in the GF (2^8) using the AES irreducible primitive polynomial that mentioned previously. The first encoded packet result as shown below.

176	234	225	123	73	71	6	5
-----	-----	-----	-----	----	----	---	---

Finally, the coding vector is attached to the coded bytes to create the final coded payload as depicted below.

41	35	111	132	176	234	225	123	73	71	6	5
----	----	-----	-----	-----	-----	-----	-----	----	----	---	---

3.2.9.3 RLNC Re-coding process

RLNC provides a unique ability called re-coding, this feature allows intermediate network nodes to re-encode the already coded packets that are received and stored at these nodes.

Consider a node has received a set of encoded packets $(C_1, X_1), (C_2, X_2), \dots, (C_M, X_M)$.

This node will choose randomly a coding vector of coefficients $z = [z_1, z_2, \dots, z_M]$ from GF(2^n) finite field to generate a new coded packet (C', X') . The linear encoding combination can be described by following:

$$X' = \sum_{j=1}^M z_j \cdot X_j$$

The encoding vector C' represents the new coding coefficients with respect to the original packets B_1, B_2, \dots, B_N . So C' coding vector is computed by

$$C'_i = \sum_{j=1}^M z_j \cdot C_{ji}$$

3.2.9.4 RLNC Decoding process

In decoding process, the decoder node receives multiple coded packets from the same or different generations to retrieve the original packets. The decoder collects the required number of coded packets to solve the system:

$$X_j = \sum_{i=1}^N C_{ji} \cdot B_i$$

Where B_i represents the unknown original packets or symbols. To recover the original packets, a linear system of equations is constructed and also can be given in a matrix form as following:

$$B = C^{-1} \times X$$

This linear system has M equations and N unknowns and it needs $M \geq N$ to be solvable and deducing all N unknown packets. So, the received coded packets should be at least as large as the number of original packets or symbols that should be recovered. The $M \geq N$ condition is sufficient only if the linear combinations are linearly independent. So, the Rank R of the solvable linear system matrix should be larger or equal to the number of original packets.

3.3 Control Plane NC architecture

In this section, we discuss in detail the SDN controller functions and the developed modules in the extended SDN-NC framework, also explain the interactions between the controller and network elements.

3.3.1 Flow Entry Management module

This Module is responsible of adding, modifying and deleting flow table entries in the data plane switches. It assigns or maps Open Flow actions with the required functions such as encoding/decoding or forwarding, and it sends them over OpenFlow messages to switch's OpenFlow agent where they should be parsed and applied. In the scenario of network coding, NC flow entries are generated and assigned based on a given multicast tree and NC scheme parameters e.g. schema type and symbol size.

When the multicast routing module decides the traffic routes from the source to destinations and which nodes would perform NC functions, a flow entry fields like in_port, output and other related fields will be filled and assigned to an action or a list of actions. The communication between this module and OpenFlow datapath agents go through TCP/TLS channels as we have mentioned earlier.

3.3.2 NC generation management module

In general, this module is to control NC generation parameters: NC scheme, generation size and Finite Field size. It supplies DPDK module generation buffer size, number of buffers and needed logical cores for each NC node.

this module is utilized to tune network coding parameters for different multicast scenarios to measure how these parameters would affect the network throughput, computation complexity and delays. Essentially, Flow Entry management module carries DPDK

parameters to OpenFlow switches on customized OpenFlow messages as explained early in OpenFlow module section.

3.3.3 Network management Module

This module provides SDN controller the capability to maintain the global view of the network, and obtaining up-to-date information about the network status. It consists of two components: Topology discovery and statistics, and that to help the controller to manage routing and monitor the existing network nodes e.g. switches and routers.

3.3.3.1 Topology Discovery

SDN discovery module uses Link Layer Discovery Protocol (LLDP) to allow Ethernet nodes to advertise their capabilities and links information. This information is stored in database and analyzed to build the network graph and recognize the nodes interconnections. In every discovery cycle, the controller sends OpenFlow Packet-out messages to switches, and each message contains the switch ID and port ID to be forwarded through to the next hop. Once the next switch received the packet it will be sent back to the controller via OpenFlow Packet-in message. This process is repeated for every switch in the network and it is performed periodically in a fixed interval to keep the links information updated. Figure 3.21 illustrates the SDN network discovery mechanism.

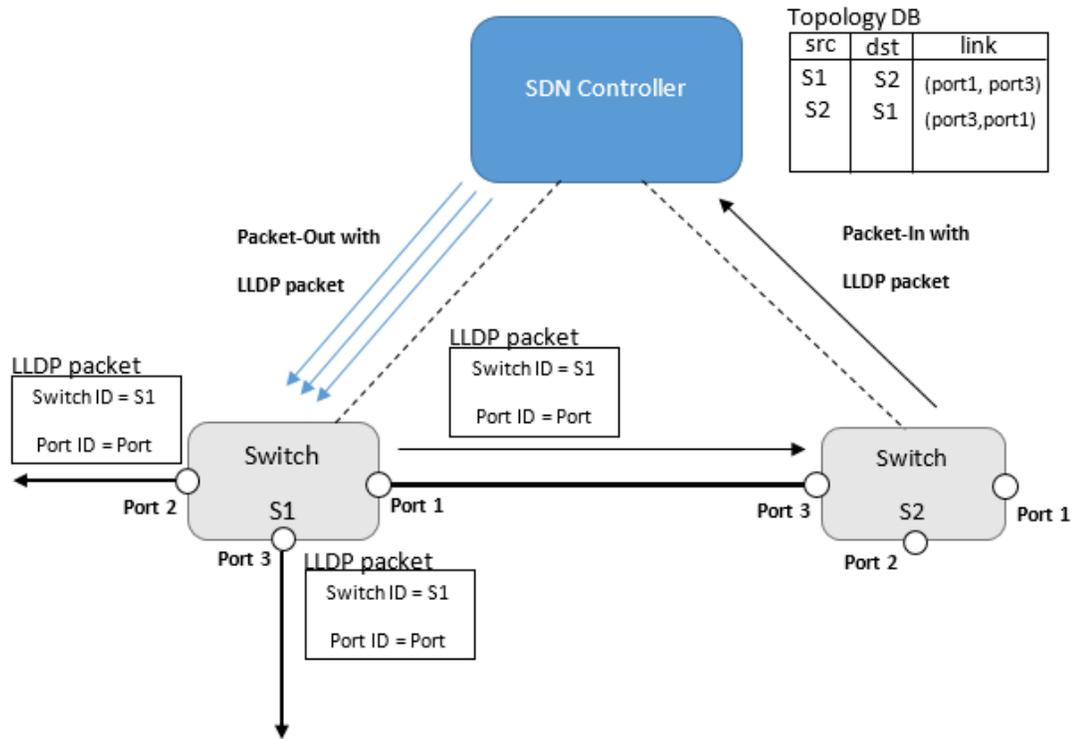


Figure 3.21 SDN Topology discovery mechanism

3.3.3.2 Statistics

Statistics service is implemented to provide more insights of SDN network behavior. it collects the statistical data from all connected OpenFlow enabled nodes. There are many types of statistics can be collected via OpenFlow protocol as listed below:

1. individual Flow Statistics
2. Aggregate Flow Statistics
3. Flow Table Statistics
4. Port Statistics
5. Group Statistics
6. Meter Statistics

7. Queue Statistics
8. Flow Table Features
9. Group Features

In our implementation, we use individual and aggregate flow statistics to measure the network bandwidth utilization and NC performance, Queue statistics to monitor the status of encoding and decoding buffers in NC nodes, Port statistics to support the controller in calculating the nodes centrality and monitoring the connectivity between them, and Flow Table Features statistics to identify the switches capabilities.

3.3.4 Multicast-Multipath Network Coding Aware Routing Module

This module is the brain of the SDN controller, it computes the data flows in the network from the source to designated multicast group members and take routing decisions based on the network capacity and traffic behavior. It automates the configuration of OpenFlow table entries in the enabled switches via flow entry management module. Routing module also discovers the network bottlenecks and select the minimum number of nodes to perform network coding functions. We can divide this module functions to the following:

- 1- Finding the shortest routes to destinations.
- 2- Measure network metrics.
- 3- Manage flow table entries.
- 4- Selecting the coding intermediate nodes.

Chapter 4

RLNC-SDN Framework Validation

4.1 Overview

In this chapter, we discuss the validation of the proposed SDN-RLNC components and define in detail the implemented functions and processes interactions.

4.2 Centrality Computation

One of the main objectives of the developed RLNC-SDN framework is to reduce the network coding computation complexity by minimizing the number of intermediate encoding nodes. So, we proposed the network centrality as a technique to discover the network coding candidate nodes.

4.2.1 Degree Centrality

Degree centrality is a simplest centrality measurement that characterize the highest centralized nodes in the network and help the SDN controller in the election process of the intermediate network coding nodes. The degree is the number of incident links or edges upon a node in a network or the number of nodes at distance one.

In the case of directed networks, there are two types of degree centrality, In-degree and out-degree according to the direction of tied links to that node.

For a given graph $G: = (V, E)$ with V vertices and E edges the degree centrality of the graph defined as in equation (4.1):

$$C_D(G) = \frac{\sum_{i=1}^{|V|} [C_D(v^*) - C_D(v_i)]}{(V-1)(V-2)} \quad (4.1)$$

let v^* with height degree in graph G .

For an example, we computed the degree centrality of the butterfly topology in Fig 1.2.

As this is a directed graph, we are interested to select a centralized node that receives more innovative packets and increase the decoding probability in sink nodes. Thus, we computed the in-degree value of each node in the graph as shown in following table:

Node	Degree	type
1	0	Source
2	1	Intermediate
3	1	Intermediate
4	2	Intermediate
5	1	intermediate
6	2	Sink
7	2	Sink

So, the butterfly network in-degree centrality:

$$C_{d_{in}} = \frac{(2-0)+(2-1)+(2-1)+(2-2)+(2-1)+(2-2)+(2-2)}{(7-1)(7-2)} = 0.16667$$

The intermediate nodes are 2, 3, 4 and 5. The node 4 is the node with highest in-degree centrality value.

4.3 Shortest Path-tree Computation

When the network graph is constructed by topology discovery module, SDN controller starts finding the shortest path-tree between the single-source node and multicast receiver nodes. In network coding aware routing module, Dijkstra algorithm [26] is applied to build the shortest path from the source to multicast receivers with the feed of network coding candidates that are selected according to the centrality analysis as we have discussed early.

The following mechanism is proposed to compute the shortest multicast tree between source and each sink node with consideration of the network coding nodes:

- 1- First step, discover all possible disjoint paths between source and sink nodes.
- 2- Compute the shortest path of each pair (source, sink) via Dijkstra algorithm.
- 3- Check if at least one coding node candidate is found in at least one of possible disjoint paths.
- 4- When a coding node is detected in a path, and the path is not previously visited, add the path to a list.
- 5- Repeat these steps for each (source, sink) pairs.
- 6- Finally, construct the path-tree table to be delivered to SDN controller's flow management module that responsible to create the corresponding OpenFlow table entries of each node.

When this mechanism is applied to butterfly scenario the following graphs are generated.

Figure 4.1 shows all possible disjoint paths between the source and sink nodes.

The solid line routes represent the shortest paths between the source node 1 and sink nodes 6 and 7 respectively. The dotted route represents the other possible disjoint path for both sink nodes as shown in graphs (a) and (b) respectively.

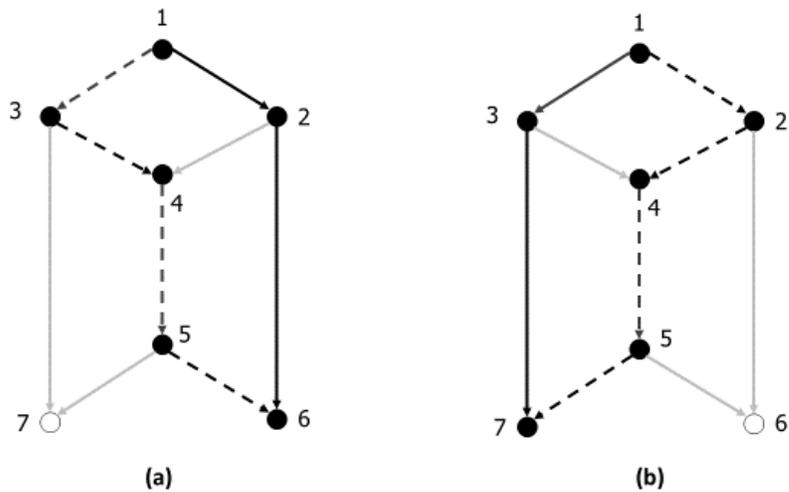


Figure 4.1 butterfly shortest disjoint paths

As we have found early, the intermediate node 4 is the highest centralized node and has been designated to be a RLNC encoding node. also, it is part of the second disjoint path between the source and both sink nodes 6 and 7.

The final constructed path-tree between the source node 1 and multicast receivers 6 and 7 is illustrated in figure 4.2.

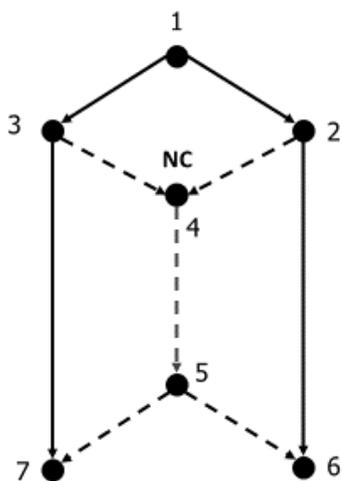


Figure 4.2 the computed butterfly path-tree via Dijkstra algorithm

4.4 Flow Computation

The number of network coding nodes should be dynamically adjusted with respect to the network traffic variable behavior, we cannot solely depend on the centrality characterization to identify the network coding nodes in the network. So, we need to take the network traffic into account to adaptively enhance the selection process of coding nodes with minimum network complexity overhead. We proposed to a technique which utilizes the SDN controller's statistics module to get more insights into the network traffic and analyze the network bottlenecks then enable network coding functions in the affected nodes. The applied mechanism is described in the following steps:

- 1- SDN controller sends a flow statistics request to all intermediate OpenFlow enabled switches between source and sink nodes as depicted in Fig 4.3.

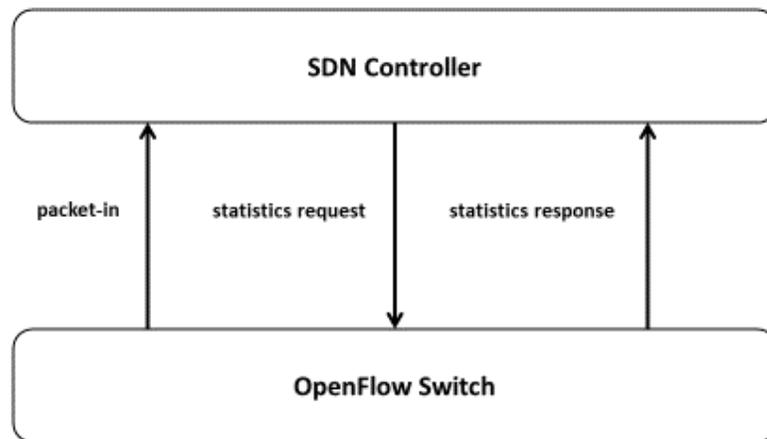


Figure 4.3 OpenFlow interactions of the statistics module

- 2- Each switch responds with flow statistics reply message that provides accumulated ports TX and RX bytes counters and the speed of switch ports.

- 3- Iterate through each port to get the packets dropping rate of incoming and outgoing traffic and save data with a timestamp as shown in Fig 4.4.

Switch ID	Port No.	Rx Counter	Tx Counter	Rx packets drop rate	timestamp
S1	1	56333 bytes	23123 bytes	5	1543576340
S1	2	76532 bytes	31870 bytes	120	1543576340
S2	1	1120 bytes	983 bytes	0	1543576340
S1	1	94322 bytes	53123 bytes	8	1543576370

Figure 4.4 OpenFlow statistics database

- 4- In a defined time interval, the previous steps will be repeated to update the switches list.
- 5- Compute the links bandwidth utilization and compare the dropping rate counters with a threshold value.
- 6- Build a list of switches that have highest packets drop rate and ports with highest bandwidth utilization.
- 7- If one of the intermediate switches is included in the list also part of the shortest path-tree nodes, mark it as network coding node.
- 8- Send a configuration OpenFlow message to the corresponding switch via flow entry management module to enable RLNC functionality and DPDK buffers.

To compute the switch link bandwidth utilization, we applied the following formula:

$$Bandwidth\ Utilization = \frac{\frac{\text{port counted bytes}}{\text{elapsed period}}}{\text{port speed}} \quad (4.2)$$

4.5 Packet Encoding and Decoding Process Verification

In this section, we will discuss the packet manipulation process in the enabled network coding switch. We take a single UDP packet as an example to simplify the process steps.

4.5.1 Packet Encoding Process

Iperf traffic generator [27] creates by default a 1470 bytes UDP payload. The UDP header size is 8 bytes (4 fields of 2-bytes). The total UDP datagram size will be 1478 bytes. Including 20-bytes IPv4 header, the total IP packet size will be 1498 bytes. This IP packet is encapsulated in Ethernet frame with 14 bytes header. The total Ethernet frame size is 1512 bytes as depicted in Fig 4.5.

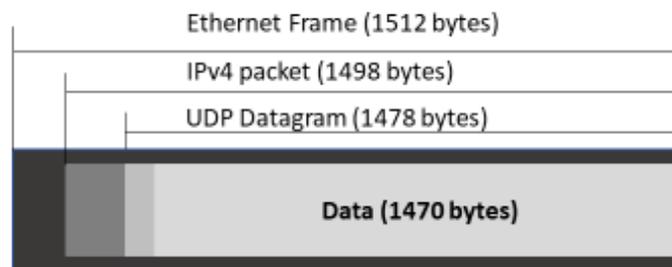


Figure 4.5 UDP Ethernet frame structure

In our network coding application, six Ethernet frames are encoded as a one RLNC generation so typically the generation size is 9072 bytes as shown in Figure 4.6.

The type of Galois Field, symbol size and number of symbols are the main RLNC encoder and decoder parameters that should be configured according to the size of RLNC generation. In our RLNC implementation, we applied network coding on GF (2^8) with 512-bytes symbol size.

The minimum number of symbols per generation is calculated as following

$$\text{number of symbols} = \frac{\text{generation size}}{\text{symbol size}} \quad (4.3)$$

We found the minimum number is 18 symbols per generation. To increase the decoding probability, we added two redundancy symbols [28] to be 20 symbols for each generation.

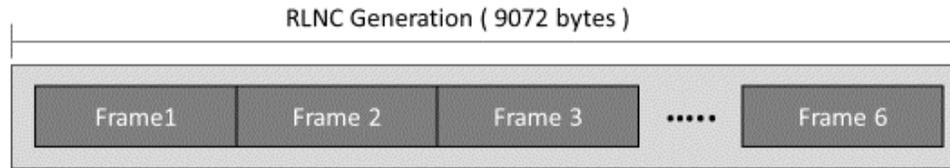


Figure 4.6 RLNC generation of six Ethernet frames

The summary of RLNC coding parameters and design consideration as follows:

Finite Field type	GF (2^8)
Symbol size	512
Number of symbols	20
Generation Size	Minimum is 9072 bytes and maximum 10240 bytes
RLNC scheme	Non-systematic

To generate a UDP traffic, Iperf tool [27] is used which creates a UDP packet that includes a random data using a one-line command:

```
#sudo iperf -u -c 10.0.0.2 |
```

The generated packet destination IP address is 10.0.0.2 to 5001 UDP port. Figure 4.7 shows an sample of single original UDP packet.

0000	0e b5 3b f8 c5 a0 b2 98 63 cc d2 0c 08 00 45 00	...;.....c.....E
0010	05 da a1 5f 40 00 40 11 7f b1 0a 00 00 01 0a 00	..._@.@.....
0020	00 02 be 2a 13 89 05 c6 19 da 00 00 02 59 5c 01	...*.....Y\
0030	af f8 00 07 73 c3 00 00 00 00 00 00 01 00 00	...s.....
0040	13 89 00 00 00 00 00 10 00 00 ff ff fc 18 36 3767
0050	38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33	89012345 67890123
0060	34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39	45678901 23456789
0070	30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35	01234567 89012345
0080	36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31	67890123 45678901
0090	32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37	23456789 01234567
00a0	38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33	89012345 67890123
00b0	34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39	45678901 23456789
00c0	30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35	01234567 89012345
00d0	36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31	67890123 45678901
00e0	32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37	23456789 01234567
00f0	38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33	89012345 67890123
0100	34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39	45678901 23456789
0110	30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35	01234567 89012345
0120	36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31	67890123 45678901
0130	32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37	23456789 01234567
0140	38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33	89012345 67890123
0150	34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39	45678901 23456789
0160	30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35	01234567 89012345
0170	36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31	67890123 45678901
0180	32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37	23456789 01234567
0190	38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33	89012345 67890123
01a0	34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39	45678901 23456789

Figure 4.7 Original UDP packet that generated via Iperf tool

When six UDP packets are received and buffered at the encoder switch port a vector of coding coefficients is generated as illustrated below:

```
symbol_coefficients_after_write_symbol:
C: 93 78 133 104 78 182 151 230 74 212 131 218 7 151 145 160 96 164 199 115
```

The original packet will be encoded using those coefficients as explained previously in chapter three and generated coded payload is as shown below:

```
symbol_data_after_write_symbol:
0000 7e 78 04 ae 72 5a 05 87 d6 78 5b f8 87 f6 c7 5c ~x..rZ...x[....\
0010 06 77 09 0c 65 b6 cc 73 f7 8d f4 80 e6 3f b0 78 .w..e..s.....?.x
....
0200

('packet size', 516)
zone:rank_before_decode,message:Rank = 15
```

The size of the encoded packet including the coefficients is 516 bytes. Then, a NC header is added to the packet and encapsulated in Ethernet frame with special type id 0x8877 the total NC packet size is 544 bytes. As we have noticed the coded packet is 35% less than the original Ethernet frame size. We discovered that the original six packets generation is remapped into 20 symbols of 512 bytes each and each encoded symbol treated as single coded packet packed with twenty coefficients. Figure 4.8 shows a sample of the generated NC Ethernet frame.

0000	6e e0 b4 7a c6 29 9a 44 54 5c bb 7d 88 77 00 00	n..z.)D T\}.}w..
0010	00 00 00 00 02 04 00 7a 00 00 00 01 00 3d e2 82Z=..
0020	f3 3b 77 b2 02 09 96 42 5e 54 17 ac 1f 8d 4c ca	.;w....B ^T....L.
0030	1a 06 3f 55 44 05 1e 22 3a 4c 0c 0d 5d b2 0e 04	..?UD..":L..]...
0040	04 06 a4 d9 15 8e 0f cc 0e 57 04 04 0c 0c 5c 3e}w....\>
0050	95 bb fb f0 ee ee 3a 3b 3e 3f 3a 3b 36 36 30 31; >?::;6601
0060	29 b2 3e 3f 3a 3b 36 38 72 71 c5 c4 c2 27 0c 0c)..>?::;68 rq...'. .
0070	0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08
0080	0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e
0090	00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a
00a0	0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00
00b0	08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c
00c0	0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08
00d0	0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e
00e0	00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a
00f0	0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00
0100	08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c
0110	0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08
0120	0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e
0130	00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a
0140	0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00
0150	08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c
0160	0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08
0170	0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a 0c 0c 0e 0e
0180	00 00 08 08 0a 0a 0c 0c 0e 0e 00 00 08 08 0a 0a
0190	3c 3d 3c 3d 34 35 3e 3f 32 33 3c 3d 3c 3d 34 35	<=<=45>? 23<=<=45
01a0	3e 3f 32 33 3c 3d 3c 3d 6a e4 b8 76 c0 2f 90 4e	>?23<=<= j..v./..N
01b0	50 58 bf 79 04 0c 43 06 0f d0 31 6d 44 04 4c 1d	PX.y..C. ..1mD.L.
01c0	ed a1 00 0a 04 05 0e 04 0c 0e ae d3 19 83 01 c2
01d0	1d de 0c 0c 06 03 50 3d d9 f5 04 0f f1 13 06 06P=
01e0	0a 0a 04 04 04 05 0c 0c 15 8f 0a 0a 04 04 04 0b
01f0	10 9d 7f b0 04 0e 9e 40 50 58 b7 71 0e 06 4f 0a@ PX.q..O.
0200	01 de 31 67 4c 0c 46 17 bf 76 88 4d f6 19 aa 7f	..1gL.F. .v.M....
0210	6a 61 29 93 2d be 70 f7 26 3b 0b 5d 7a 39 2c 11	ja).-..p. &;.]z9,.

Figure 4.8 Encoded packet encapsulated by NC Ethernet frame

4.5.2 Packet decoding process

When a single encoded packet traverses the network and reaches the decoding switch port, it will be decapsulated and the network coding header will be inspected to forward the packet to the corresponding generation buffer to be prepared for decoding process.


```

000 U:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
001 U:  0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
002 U:  0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
003 U:  0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
004 U:  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
005 U:  0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
006 U:  0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
007 U:  0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
008 U:  0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
009 U:  0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
010 U:  0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
011 U:  0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
012 U:  0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
013 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
014 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
015 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
016 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
017 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
018 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
019 U:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

```

Figure 4.10 RLNC decoder status matrix of full recovered NC generation

Chapter 5

Experimental Setup and Implementation Tools

5.1 Experiment Environment and Tools

5.1.1 Mininet

We have implemented our extend SDN-NC framework in Mininet. Mininet [29] is a network emulation environment that able to create virtual hosts, OpenFlow switches, controllers and links.

It supports arbitrary custom topologies; and provides flexible and simple python APIs to run network-wide tests. Mininet uses process-based virtualization to run up to 4096 hosts and switches on a single OS Linux/Unix Kernel. It provides the individual processes with separate network interfaces, routing and ARP tables.

Mininet provides a unique connectivity interface for SDN controller to interact with all emulated OpenFlow switches. Mininet is capable to connect virtualized data elements to remote controller or create a standard local controller in the same emulation environment.

In Mininet we can create a python configuration file to characterize the emulated network scenario and configure the virtualized network elements with the desired parameters such as CPU limit and link delay. Mininet CLI is a command line tool that can perform ping tests between the nodes and display many useful information such as links status, and initialize the shell terminals for the nodes. In each node terminal, we can perform common Unix/Linux commands and execute scripts into these nodes.

We have installed and configured Mininet emulator based on the following specifications:

- **Hardware specifications:**
 - Intel core i7 2.8 GHz 4 cores CPU Dell Workstation.
 - 16 GB DRAM.
 - 100Mbps Ethernet network Interface.
- **Operating systems:**
 - Xen server v7.0 virtualization host.
 - Ubuntu server 14.04 64-bit for virtual machines.
- **Software:**
 - Mininet v2.2.0
 - Python 2.7

5.1.2 Wireshark

Wireshark [30] is a well-known opensource network analyzer, it is able to capture network traffic from live virtual or physical network interfaces in promiscuous mode or from already-captured packets file. It can dissect, parse and filter many different network protocols.

In Mininet, we use Wireshark to capture multicast network traffic and encoded packets for different SDN network scenarios to analyze and measure our desired performance metrics as we will discuss later in detail.

Wireshark has wide range of statistics tools that able to show different analytics like delay, I/O graphs, average throughput...etc.

5.1.3 RYU Controller

RYU is an opensource SDN controller [31] written in python programming language, it provides well-defined APIs and software components that simplify the development of network and control applications. RYU controller supports OpenFlow 1.3 control protocol and many other protocols. We have utilized the RYU built-in components such as statistics and topology discovery modules in our SDN-NC framework as we have mentioned in chapter three. We have implemented our customized multicast routing protocol as a network application that runs on top of RYU framework. This application is responsible to do number of major operations as described below.

1. Identify the network graph or topology matrix. RYU controller has a readymade topology discovery library (`ryu/topology/api.py`) where can be utilized for this purpose. the main library functions are:
 - **Get_all_switch():** this function discovers all switches that connected to the controller.
 - **Get_all_link():** this function discovers all links information for switches and hosts.
 - **Get_all_host():** this function discovers all hosts that connected to OpenFlow switches within the network.
2. Compute the shortest paths between the source node and the receivers. For this purpose, there are different algorithms can be used such as Dijkstra, Bellman-ford

and Floyd–Warshall. Each algorithm has cons and pros in term of the number of connected nodes and links, and the required computation resources to get the optimal path or paths. We use a third-party python library known as NetworkX [32]. NetworkX is a python software package for creation, manipulation, and study of the structure, dynamics and function of complex networks. This library includes functions to find the shortest path between two vertices or more in a network graph that fed by the RYU topology API. The main functions we apply in our implementations are:

- **all_shortest_paths(G[, source, target, weight])** : compute all shortest paths in the graph between a source and a target node.
- **single_source_shortest_path(G, source[,cutoff])** : compute the shortest path between a single source to all reachable destinations of that source.
- **all_pairs_shortest_path(G[, cutoff])**: compute all shortest paths between all nodes in the network graph.
- **has_path(G, source, target)**: Return True if the graph has a path from a source to sink node, False otherwise.
- **Is_directed(G)**: check the network topology is it directed or cycled.

G = network graph matrix.

Source = source node.

Target = destination node or nodes.

Cutoff = search depth.

Weight = the link weight can be specified based on different parameters like link loss, delay or bandwidth.

3 Compute the nodes centrality; when the shortest path is specified, the controller discovers the high centralized nodes in that path and calculate the required parameters (α , β), to identify the suitable intermediate coding nodes as explained previously in CNCNS algorithm [20]. To compute the centrality, we utilize the centrality functions in NetworkX library as listed below:

- **load_centrality(G,W,cutoff[,options])** : Return the load centrality values for nodes in the graph. The weight parameter represents the load or utilization of the edges or links in the graph.
- **degree_centrality(G)**: Compute the degree centrality for graph nodes.

3. Generate and assign network coding flow entries into the correspondent OpenFlow switches. RYU controller contains OpenFlow 1.3 API functions that can help us in configuring and sending OpenFlow messages to the supported switches.

- **ryu.ofproto.ofproto_v1_3_parser.OFPFeaturesRequest(datapath)**: creates a features request message to establish a connection between the controller and a switch , also discovers the switch basic information and capabilities. After the connection is established, the switch responses with FeatureRes message the response example is depicted as following:

```

{
  "OFPSwitchFeatures": {
    "auxiliary_id": 99,
    "capabilities": 79,
    "datapath_id": 9210263729383,
    "n_buffers": 0,
    "n_tables": 255
  }
}

```

Figure 5.1 OpenFlow Feature Response data

- **ryu.ofproto.ofproto_v1_3_parser.OFPFlowMod(datapath,match,instructions,table_id,priority,buffer_id,flags):** this function is responsible to configure the switch flow table and add the required flow entries into it. The match parameter includes the matching fields to be compared against packet header fields. Instructions parameter should contain the list of actions that have to be applied on the matched packet.
4. Collect flows and ports statistics and monitor the links status. RYU controller provides API functions that retrieve these statistics from all connected switches or dataplane elements. The following are the main functions:
- **ryu.ofproto.ofproto_v1_3_parser.OFPFlowStatsRequest(datapath,flags,table_id[,options]):**The controller uses this function to send statistics query message to the switch. The following is an example of a response message from the switch.
 - **ryu.ofproto.ofproto_v1_3_parser.OFPPortStatsRequest(datapath,flags,port_no):**the controller uses this function to request information about the switch port statistics. The following the switch response for the query:

```

{
  "OFPPortStatsReply": {
    "body": [
      {
        "OFPPortStats": {
          "collisions": 0,
          "duration_nsec": 0,
          "duration_sec": 0,
          "port_no": 7,
          "rx_bytes": 0,
          "rx_crc_err": 0,
          "rx_dropped": 0,
          "rx_errors": 0,
          "rx_frame_err": 0,
          "rx_over_err": 0,
          "rx_packets": 0,
          "tx_bytes": 336,
          "tx_dropped": 0,
          "tx_errors": 0,
          "tx_packets": 4
        }
      }
    ]
  }
}

```

Figure 5.2 OpenFlow port statistics response data

5.1.4 OpenvSwitch

OpenvSwitch (OVS) [33], is a multilayer opensource software switch, it supports OpenFlow protocol and various networking protocols such as VLAN, VXLAN,sFlow, NetFlow ...etc. this switch can reside in a server, virtual machine or any general purpose hardware. Using OVS, we can develop, implement and test new networking functions, and extending OpenFlow protocol to more added features. We have included network coding capabilities into OVS kernel module and extended the OpenFlow protocol to enable the configuration of NC functions.

OVS consists of the following main components:

- **Ovs-switchd:** a daemon that implements the switch in Linux environment and enables it for flow-based switching.
- **Ovsdb-server:** a lightweight database server to store the switch configurations.
- **Ovs-dpctl:** a tool for configuring the switch kernel module.

In our implementation, we have replaced the standard OVS switch with our NC enabled switch, and configured Mininet environment to use the extended one in all emulated switching nodes.

Recently, DPDK library has been included in OVS to accelerate packet processing. DPDK library is utilized to enhance the encoding/decoding process and optimize the consumed hardware resources.

5.1.5 RLNC Libraries

For RLNC implementation, Kodo libraries [34], are chosen as a framework to develop custom network coding applications. Kodo provides simple and efficient APIs and support various network coding codecs e.g. standard RLNC, sparse RLNC, systematic RLNC. Kodo libraries are implemented in many programming languages such as C/C++, python and Java. In SDN-NC framework, we use Kodo libraries in C and we have implemented coding, recoding and decoding functions as customized features in OpenvSwitch kernel module.

5.2 Performance metrics

The performance metrics that used to evaluate our extended NC-SDN framework in Mininet scenarios are the following:

5.2.1 Throughput

Throughput is the rate of data that transferred successfully over a communication channel. In our implementation, we measure the throughput for both encoded and decoded traffic at the receiver nodes.

5.2.2 Delay /Latency

End-to-End packet delay is the total time between sending a packet from a source and receiving it successfully at the receiver. In our case, the delay is the time between sending a packet at the source node and its successful decoding at the receiver. In addition, we calculate encoding and decoding average delays.

5.2.3 CPU Utilization

To evaluate the computation overhead of networking coding, we measure the CPU load of each encoder and decoder node in the network. Each node inside Mininet environment runs

as a normal CPU process, so it can be easily monitored via any Linux resources usage viewer such as Top utility. To measure the CPU utilization for each node, we wrote a script to monitor Mininet processes every second and record the values in a CSV file.

5.3 The Experimental Steps

1- Run the Mininet topology configuration file to initialize the topology switches, hosts and connecting them via virtual links.

2- Initialize Ryu SDN controller and assign the network app to Ryu app-manager to connect the controller with the initialized OpenFlow switches to be able to control them and recognize the overall network activities.

3- Use ping tool to ensure the reachability from the source node to each receiver node.

4- Ryu network app configures the designated coding/decoding switches with the required NC configuration parameters.

5- Initialize Wireshark to listen to the desired ports to collect the network traffic and make it ready for the analysis.

6- Run Iperf tool to generate a UDP traffic from the source node to the end destinations.

7- Apply filters in Wireshark to differentiate between the coded packets and the original generated packets.

8- Save the collected traffic as pcap file (packets capture file) and prepare it for analysis and performance measurements using MATLAB or Excel.

5.4 SDN Multi-hop Reference Scenario

To compare our developed SDN-RLNC framework with a reference baseline SDN deployment, we have built a simple multi-hop topology on Mininet emulator and measure the performance metrics: throughput, delay and computation power as we have discussed previously.

The proposed SDN reference topology consists of two switches and two hosts (source and sink nodes) as shown in Figure 5.3.

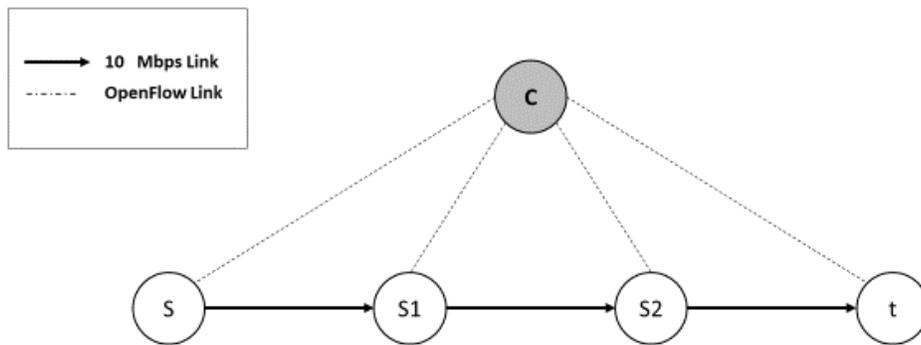


Figure 5.3 SDN multi-hop topology

The multi-hop topology is built and configured in Mininet as one source node S, two cascaded OpenFlow switches S1 and S2 and one sink node t. The switches are supervised by one SDN controller C to control traffic forwarding between source node S and sink node t via these two switches and three links of 10 Mbps bandwidth each.

A generated 5 Mbps UDP traffic, is injected from source to sink node using Iperf tool and forwarded via switches S1 and S2 to be received finally at sink node t.

The results show that the average end-to-end throughput is 4.35 Mbps and the average forwarding and propagation delay is 7.2ms. The average CPU processing power is 4%.

Chapter 6

SDN-NC Experiments and Results

6.1 Butterfly Scenario

Butterfly scenario is the most common topology in network coding theories. First, the butterfly scenario was built and initialized in Mininet SDN environment as depicted in Figure 6.1. The topology parameters were configured into a python configuration file (butterfly_topology.py). The experiment parameters are listed in Table 6.1.

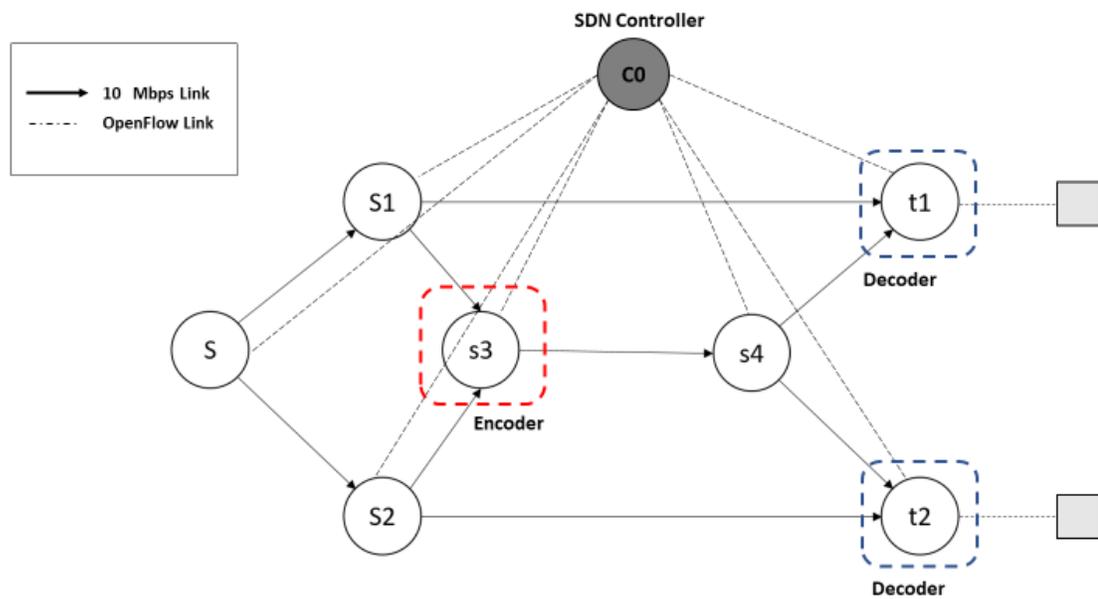


Figure 6.1 Mininet SDN-RLNC butterfly topology

Parameter	Value
Source node	S
Receiver nodes	t1, t2

Controller node	C0
Intermediate nodes	S1, S2, S3, S4
NC Encoding nodes	S3
RLNC codec type	Non-systematic RLNC
Finite Field size	2^8
Maximum symbol size	512 bytes
Number of symbols per block	20 symbols
Maximum Ethernet frame size	1512 bytes
UDP packets Generation size	6 packets
Data rates range	1Mbps to 10Mbps
Data injection Duration	10 seconds
Centrality type	Degree Centrality
Hosts IP addresses range	10.0.0.1/8 – 10.0.0.3/8

Table 6.1 Mininet SDN-RLNC butterfly configuration parameters

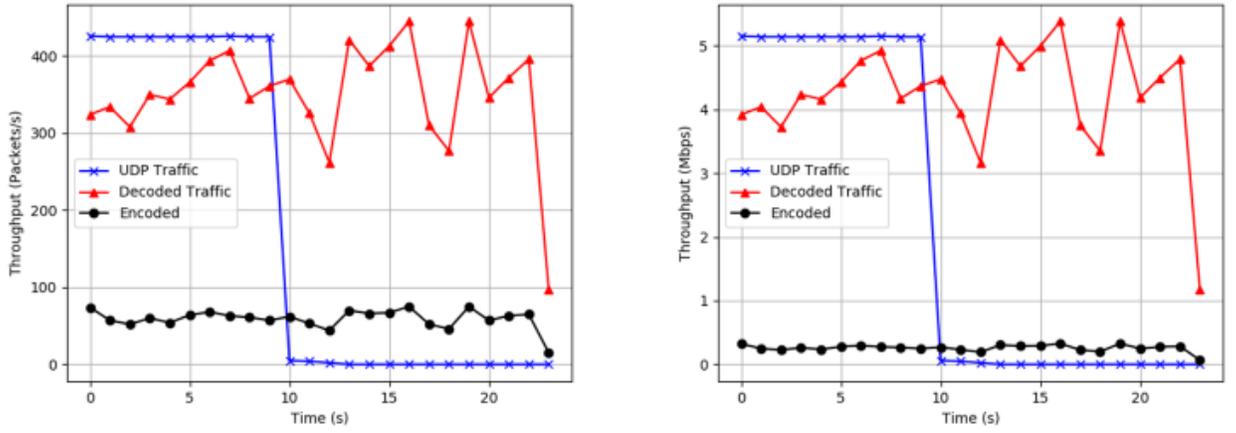


Figure 6.2 Butterfly Throughput of three traffic types: UDP, encoded and decoded

6.1.1 Throughput

Figure 6.2, shows the resulted throughput of three types of traffic: encoded, decoded and original UDP. In a sample experiment, a UDP traffic was injected in 5 Mbps data rate for 10 seconds which generated 4263 packets of 1512 bytes packet size. As we noticed, the RLNC encoding switch S3 generated in average 60 encoded packets/s in size of 544 bytes each. When the decoding nodes t1 and t2 buffers started receiving RLNC packets, the number of recovered UDP packets started increasing gradually and reached the highest rate 445 packets/s, when 15 seconds passed, the decoding rate exceeded the maximum UDP traffic throughput from the source node (425 packets/s) as reflected in the graph (a). On the other hand, the average decoding rate was 350 packets/s. The graph (b) illustrates the throughput of the same types of traffic but in Mbps unit. The average encoded traffic was 0.26 Mbps, the decoded throughput was 4.23 Mbps and original UDP rate was 5.14 Mbps.

Every six original UDP packets form a single generation to create one coded packet of size 544 bytes. To calculate mathematically the encoded packets rate as following:

$$Encoded\ Packets\ rate = \frac{Original\ UDP\ packets\ rate\ (bits/s)}{Generation\ size\ (bits)} \quad (6.1)$$

So, the maximum theoretical encoded packets rate is 70 packets/s and the actual measured encoding rate at node S3 was 60 packets/s, the reasons why this value is lower than the maximum rate; are the flow-match latency and encoding operations.

To compute the source node maximum packets rate, we use the following equation:

$$UDP \text{ packets rate} = \frac{\text{Initial data rate bits/s}}{\text{maximum packet size (bits)}} \quad (6.2)$$

The computed UDP traffic rate value was 425 packets/s, where it matched the measured UDP traffic using Wireshark tool, experimentally, it was found that the iperf tool sends data payloads (1470 bytes) in 5 Mbps data rate and the Ethernet frames are transferred at 5.14 Mbps to include the headers extra bytes.

At a sink node, the ideal decoded traffic rate can be calculated as following:

$$Decoded \text{ Packets rate} = Encoded \text{ packets rate} * Generation \text{ Size} \quad (6.3)$$

The expected value of decoded packets rate is 360 packets/s, the average encoded rate is 60 packets/s and the generation size is 6 packets. The measured result of our scenario was 350 packets/s in average. This value is less than the computed one and found that due to two main factors:

- The number of received coded packets at each sink node.
- Generation independent property between the code words of the received coded packets.

These two factors determine the rank of each generation equations matrix, and eventually effect the number of decoded packets for each time interval.

The generations matrices with low rank will be delayed because they are waiting to receive more independent coded packets of the same generation.

Throughput validation

To validate the throughput of the developed framework, multiple experiments were run by injecting UDP packets for a range of data rates. The graphs on Figure 5.3 reflect the throughput behavior when the UDP traffic rates varied from 1 to 10 Mbps with 95% confidence interval. In graph (a), we observed that the initial coded and decoded traffic throughputs are both proportionally increasing when the UDP data rates are increasing from 1 to 3 Mbps. After that, throughput of coded and decoded traffics saturated when the UDP traffic reached 4 Mbps. the resulted average saturation data rate of decoded traffic was 362 packets/s and coded traffic was 61 packets/s. The graph (b) illustrates the throughput in Mbps Unit, the average encoded throughput was 0.26 Mbps and decoded throughput rate was 4.3 Mbps.

The main factors that affect the decoding throughput rate are following:

- The rate of received coded packets at the decoder node, we found the encoding process at NC node generates a variable number of encoded packets every second, so the decoder output rate is bounded by the number of received encoded packets.
- The computation complexity and generation size to determine the recovery of each generation of the original UDP traffic.

The main factors affect the encoding throughput rate as following:

- The rate of received UDP packets at the encoder. When the UDP traffic exceeds the encoder processing capacity, the packets start stacking in the encoder memory and that impose a significant delay as described in the delay measurements section.
- The flow-match process and buffering of each generation also limiting the encoding efficiency.

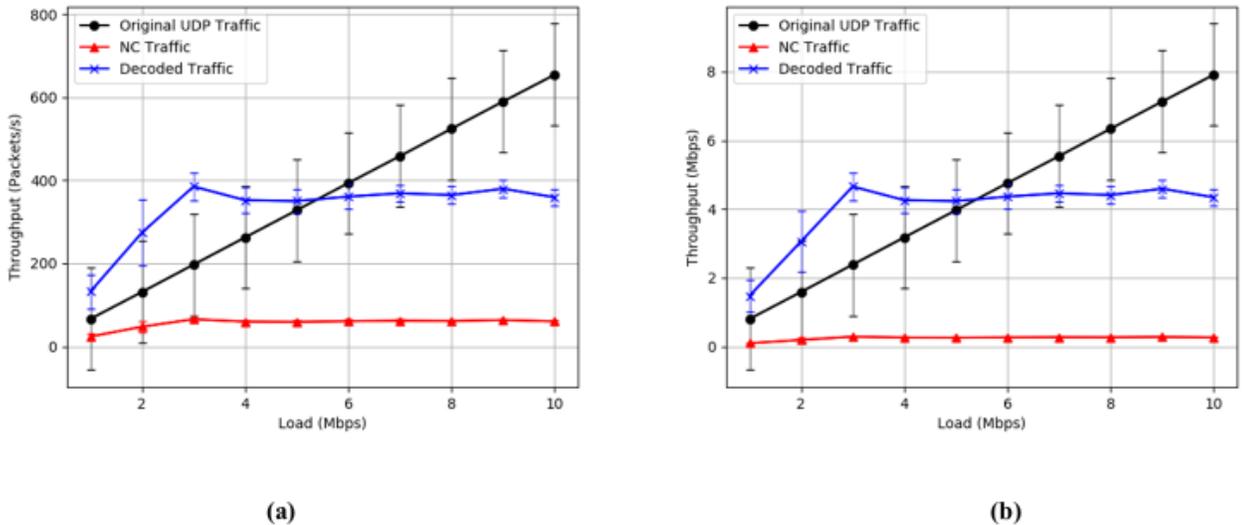


Figure 6.3 Butterfly Throughput versus UDP traffic load

6.1.2 Delay/Latency

In this scenario, we have measured the imposed delay of RLNC coding and decoding operations. We have computed the delay using the timestamp of each packet enters or exits a measured coding or decoding node. The absolute difference value between two packets timestamps provide us the duration of each packet has spent to be generated or processed.

The graphs in Figure 6.4, reflect the encoding delay versus different UDP traffic loads and 95% confidence interval. Graph (a) shows the total delay time that required to encode the received original UDP packets to new coded ones.

We found that the coding delay time is proportionally increasing with the increment of applied UDP traffic load, and the average encoding delay was 18 ms to generate a single coded packet as shown in graph (b).

Graph (b) shows the initial delay to generate a coded packet was 39 ms and then steeply decreased to stabilize around 16 ms per coded packet. The reason that the initial delay is high; the encoder has been stalled because rate of received UDP packets was less than the encoding time processing rate so the encoder was waiting more time to buffer the sufficient number of UDP packets to encode them.

In Figure 6.5, the graph (a) illustrates the total delay time to decode all received NC packets at a decoder node t1. The observed decoding delay time was also increasing when we injected the network with higher traffic loads, and that generated many encoded packets generations to be decoded. Graph (b) depicts the initial delay to decode a packet was 6.6 ms, when the load was increasing, the delay decreased to 2.6 ms in average. the average decoding delay to decode a single UDP packet was 3.13 ms.

The reason that the initial delay in the decoder did not receive the sufficient number of innovative coded packets for decoding process as we have discussed the decoding process it in chapter three.

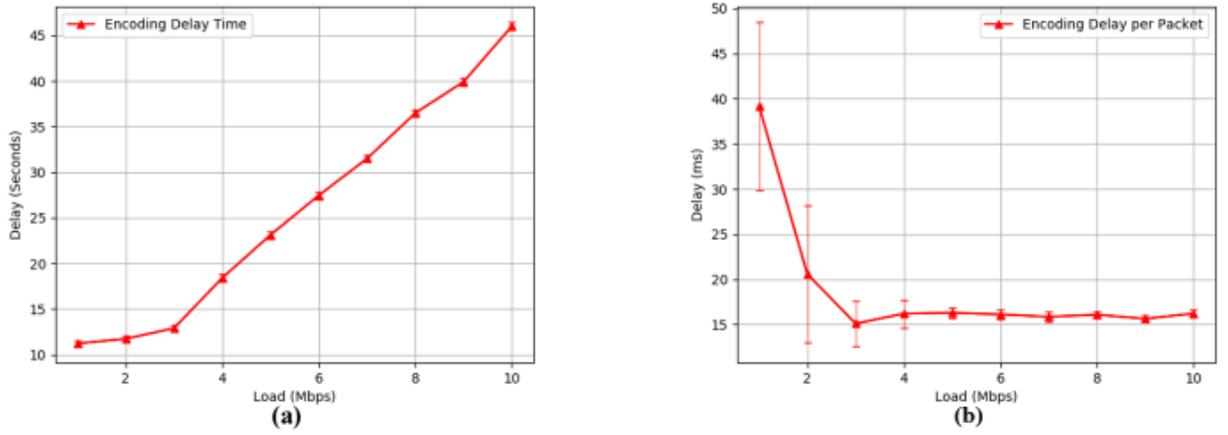


Figure 6.4 butterfly coding delay/latency versus UDP traffic load

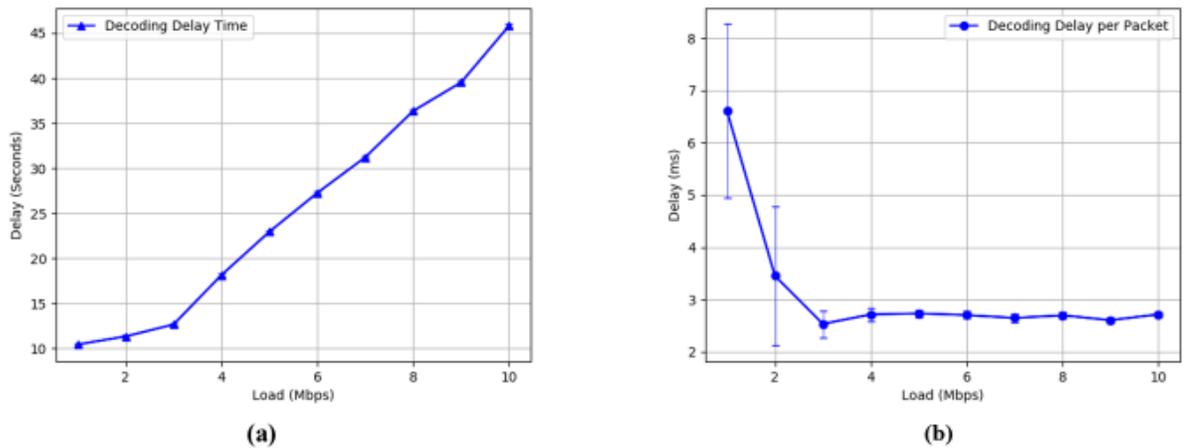


Figure 6.5 Butterfly decoding delay/latency vs UDP traffic load

6.1.3 CPU Utilization / Computation Power

To study the effect of network coding operations on OpenFlow switches, we have measured the CPU utilization for encoding and decoding nodes, and how this would affect the emulation environment (Mininet). In a sample experiment, a 5 Mbps UDP traffic was injected at the source node for 10 seconds, and the CPU usage was monitored and recorded for all encoding, decoding and Mininet processes as explained before in chapter four. The output results show that the decoding CPU utilization was increasing exponentially when the number of arrived NC packets were increasing at the receiver node.

T1 decoding node's CPU usage reached 96% of the virtualized switch CPU power. As we have assumed, the high CPU consumption is necessary to decode each RLNC generation of the received packets at higher data rates.

The decoding's process creates a thread (sub-process) to decode each generation and to reserve the not decoded UDP packets in the memory. Each thread tries to solve the system of equations when it receives a new coded packet of the same generation along with the previous not decoded packets of the same generation.

On the other hand, the encoder's CPU usage ramped up to 60% and then decreased gradually to reach 32%. We found the encoder's process kills the created threads when it completes the encoding process and deallocate the reserved memory resources.

Also, we observed the effect of network coding operations on the emulation environment itself was very minimal where CPU usage increased only 6%, because the Mininet and its emulated nodes are handled as isolated processes by the operating system. All observations are summarized on Figure 6.6.

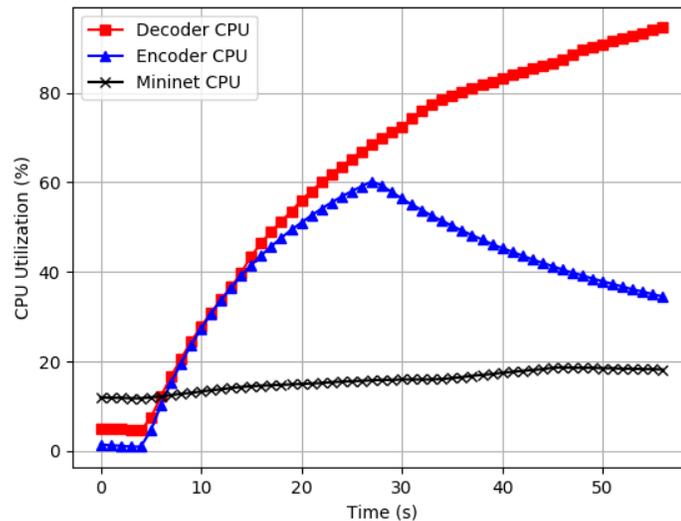


Figure 6.6 Butterfly CPU Utilization

CPU Utilization Validation

To validate the impact of RLNC operations on the machine's resources, various experiments have been run by injecting a range of UDP traffic loads (1 to 10) Mbps into the source node S. The average CPU utilization of the encoding node S3 and both decoding nodes t1 and t2 are measured for each UDP traffic load. The results show that the CPU power consumption is also increasing when the value of the injected load is increasing as shown in Figure 6.7.

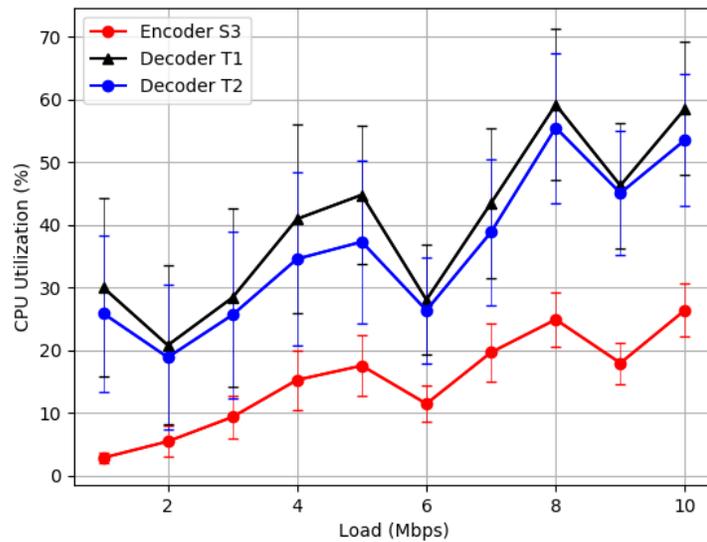


Figure 6.7 Butterfly CPU utilization versus UDP traffic load

6.2 Multicast Fat-tree scenario

To evaluate our RLNC-SDN implementation in a complex and real-like scenario, Fat-tree topology has been chosen, where it is widely used in SDN data centers [35]. The Fat-tree topology has been built in Mininet and connected the emulated OpenFlow switches with one Ryu controller as illustrated in Figure 6.8.

The fat tree topology consists of three layers of switching:

- **Core switches:** The top tier switches or routes that exchange traffic between the data center and the outside networks e.g. Internet.

- **Aggregate switches:** represent the intermediate level switches that connect the data center clusters; and aggregate the traffic to the upper core layer.

- **Edge switches:** referred to access switches that connect servers and/or storage nodes of the same cluster and exchange their traffic with upper layers.

The fat-tree topology is implemented as the configuration settings that listed in Table 6.2.

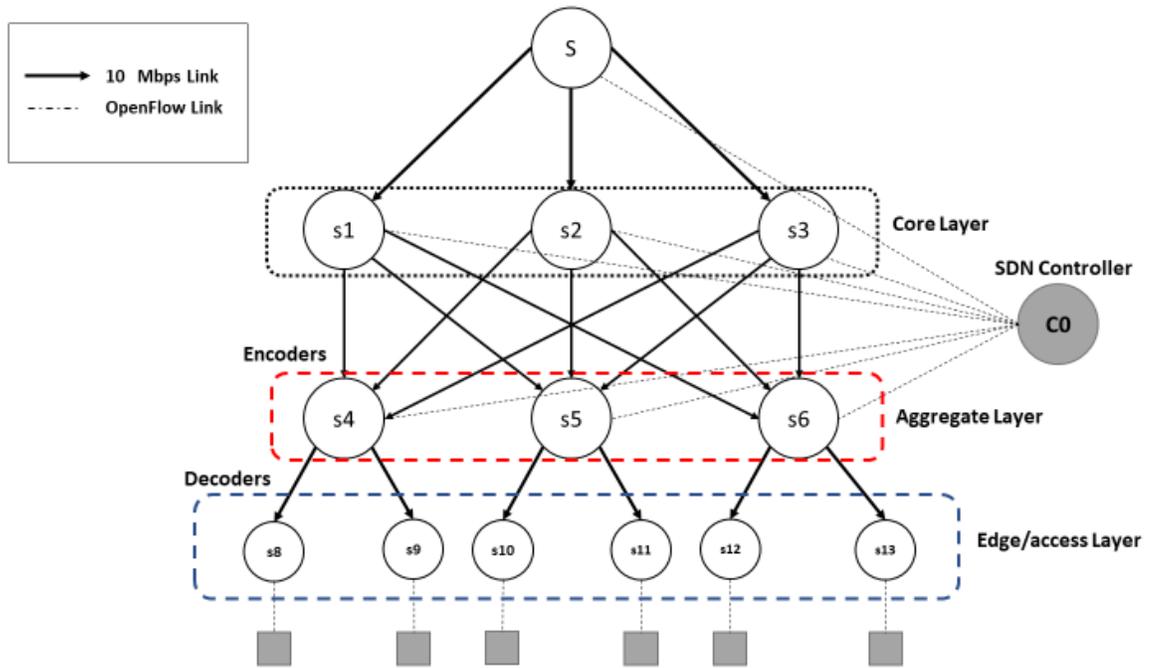


Figure 6.8 Mininet SDN-RLNC Fat-tree Topology

Parameter	Value
Source node	S
Receiver nodes	S8, S9, S10, S11, S12, S13
Controller node	C0
Intermediate nodes	S1, S2, S3, S4, S5, S6
NC Encoding nodes	S4, S5, S6
RLNC codec type	Non-systematic RLNC
Finite Field size	2^8
Maximum symbol size	512 bytes

Number of symbols per block	20 symbols
Maximum Ethernet frame size	1512 bytes
UDP packets Generation size	6 packets
data rate range	1 Mbps to 10 Mbps
Centrality type	Degree Centrality
Hosts IP addresses range	10.0.0.1/8 – 10.0.0.7/8

Table 6.1 Mininet SDN-RLNC Fat-tree configuration parameters

6.2.1 Throughput

Figure 6.9 reflects the resulted throughput of three types of traffic: encoded, decoded and original UDP in the fat-tree scenario. The same sample experiment was repeated, and a UDP traffic of 5 Mbps data rate was injected into the source node for 10 seconds which generated 4259 packets of 1512 bytes packet size each.

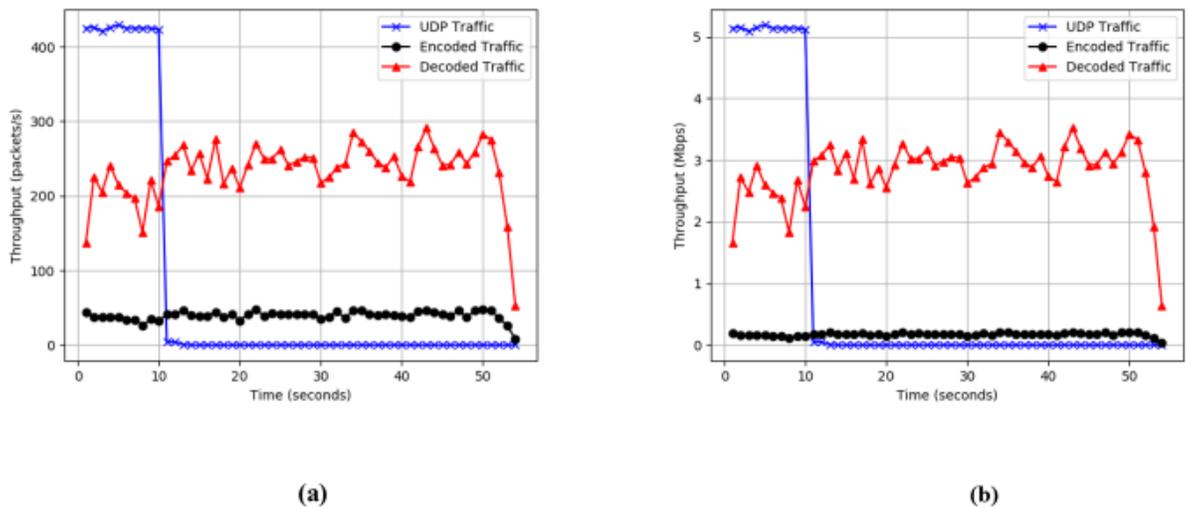


Figure 6.9 Fat-tree Throughput of three traffic Types: UDP, encoded and decoded

The average data rate of the injected UDP traffic at the source node was 425 packets/s (5.1 Mbps). Arbitrarily, the generated encoded traffic from coding node S4 was measured and the average throughput rate was 40 packet/s (0.17 Mbps), where it is lower than the average encoding rate of a single encoding node in the butterfly scenario. When we investigated the encoding throughput of the two other NC nodes (S5, S6), we found the node S5 had the same average encoding rate of S4 (40 packets/s), but S6 average rate was 30 packets/s. This observation confirmed that the number of intermediate nodes and the topology size can directly affect the encoding throughput rate.

To compute the decoding traffic rate, the equation 5.3 was applied, and the computed decoded traffic is $(40 * 6) = 240$ packets/s (2.9 Mbps). The actual decoded average throughput at decoding node S8 (connected to S4 node) was 234 packets/s (2.8 Mbps) where it is also below the computed value. So, the rate of decoded packets is bounded by the encoded packets rate and the computation process time.

Throughput validation

To validate the throughput for Fat-tree scenario, multiple experiments were run by injecting UDP packets into the source node S for a range of data loads.

In Figure 6.10, graph (a) summarizes the output encoded traffic of encoding nodes S4, S5 and S6 at each link that connected to the corresponding decoding nodes (S8, S9, S10, S11, S12, S13) as illustrated in the topology diagram, versus the applied UDP traffic loads in packets/s unit. Graph (b) reflects the same encoded throughput rates in Mbps unit.

Both graphs show clearly the encoding rate at node S6 was severely affected and the average throughput value was only 29 packets/s (0.12 Mbps) in average for all traffic loads where the average encoded throughput was 40 packet/s (0.17 Mbps) for other coding nodes

S4 and S5. This behavior has led us to investigate and study the reasons more thoroughly as will be shown shortly.

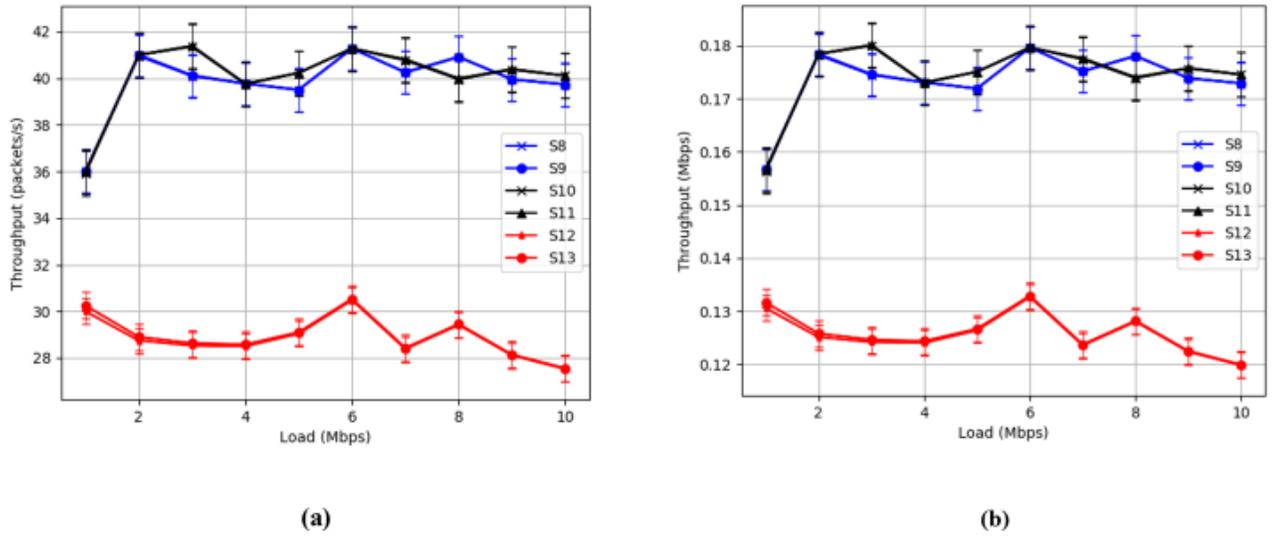


Figure 6.10 Fat-tree topology encoded throughput versus UDP traffic load

In Figure 6.11, graph (a) illustrates the throughput of decoding nodes S8, S9, S10, S11, S12 and S13 in packets/s unit. Graph(b) reflects the same decoded throughput in Mbps unit.

The results show the average throughputs of decoding nodes S8, S9, S10 and S11 were similar and stable around 234 packets/s (2.8 Mbps) across all traffic loads. On the other hand, the decoding rate of nodes S12 and S13 was fluctuating and below the average of the previous decoding nodes.

So, to investigate this behavior all nodes were monitored to identify the reasons behind it.

It was found that, when the Fat-tree SDN scenario was initialized and the UDP traffic started to flow the topology, the encoding and decoding processes started greedily consuming the CPU power of the virtual machine and that has affected the allocation of CPU resources for each node in the topology to process the received packets either for encoding or decoding. To ensure that the behavior was not topology related, the initial node

in the encoding and decoding processes was randomized; the behavior was repeated but this time different nodes were affected at each experiment, which also confirms that the large number of NC nodes requires more computation resources.

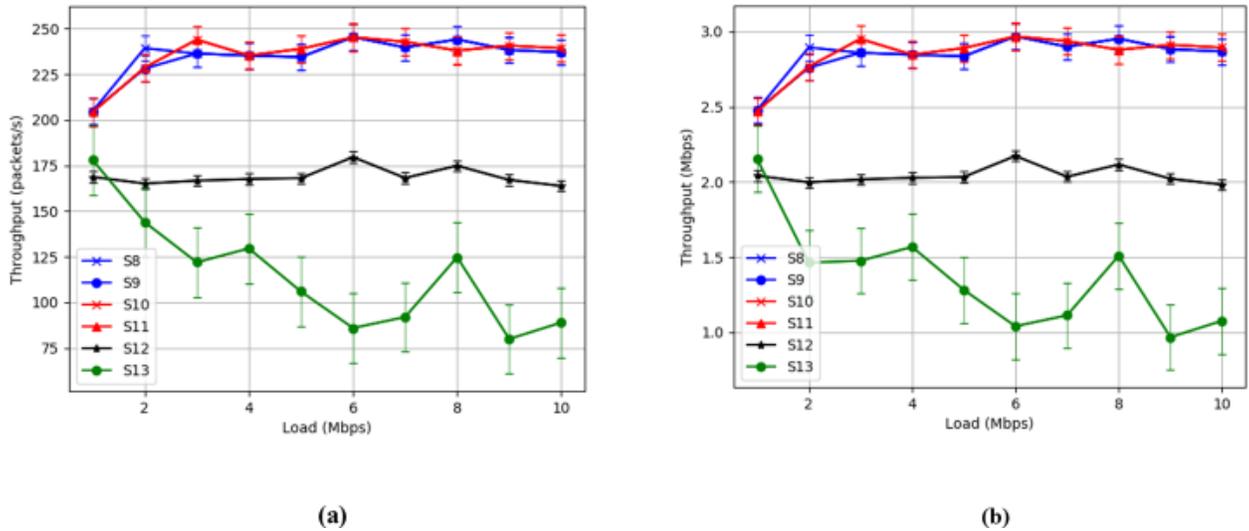


Figure 6.11 Fat-tree topology decoded Throughput versus UDP traffic load

6.2.2 Delay/Latency

In Figure 6.12, graph (a) shows the resulted the total delay/latency of encoded traffic of all encoding nodes versus the applied UDP traffic data rates at the source node S. graph(b) shows the average delay to generate a single coded packet versus the applied UDP traffic loads with 95% confidence interval.

Graph (a) reflects the proportional relation between the injected UDP traffic and the time required to encode the received UDP packets at the coding nodes S4, S5 and S6. As shown from the graph as we increased the injected traffic load the total delay time grew longer, reflecting a linear relationship.

In graph (b) the average delay of a single coded packet is observed throughout the topology instead of the total average. As described above in the throughput validation of the encoded nodes, the average delay per packet is also affected by the limited computation resource,

and since the throughput and delay are correlated, the same behavior was observed through the encoded nodes.

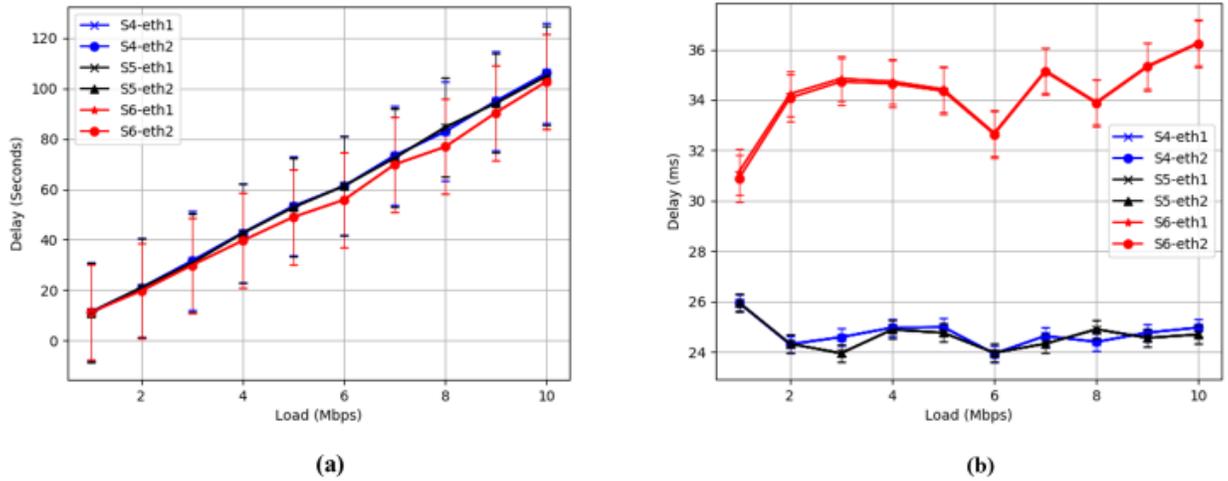


Figure 5.12 Fat-tree traffic latency of encoded traffic versus UDP traffic load

In Figure 6.13, graph(a) depicts the total average delay time to recover the applied UDP traffic at the decoding nodes with 95% confidence interval. Graph(b) shows the average delay to recover a single UDP packet versus various UDP traffic loads for each decoding node.

It was found that nodes S12 and S13 were affected by the instability of the throughput of the encoding node S6, and in graph (a) although the relationship between the injected traffic load and the total average delay looks like Figure 5.12, graph (a), except that node S13 does not follow the same trend due to shortage of allocated CPU resources.

In graph (b) where the average delay of a single packet is depicted, it is found that all the decoding nodes except for nodes S12 and S13 could utilize the limited CPU resource and process the packet with short delay even with the increased load, but nodes S12 and S13 suffer very long delay time due to the instability of encoding node S6 providing the packet as well as the limited computation resources.

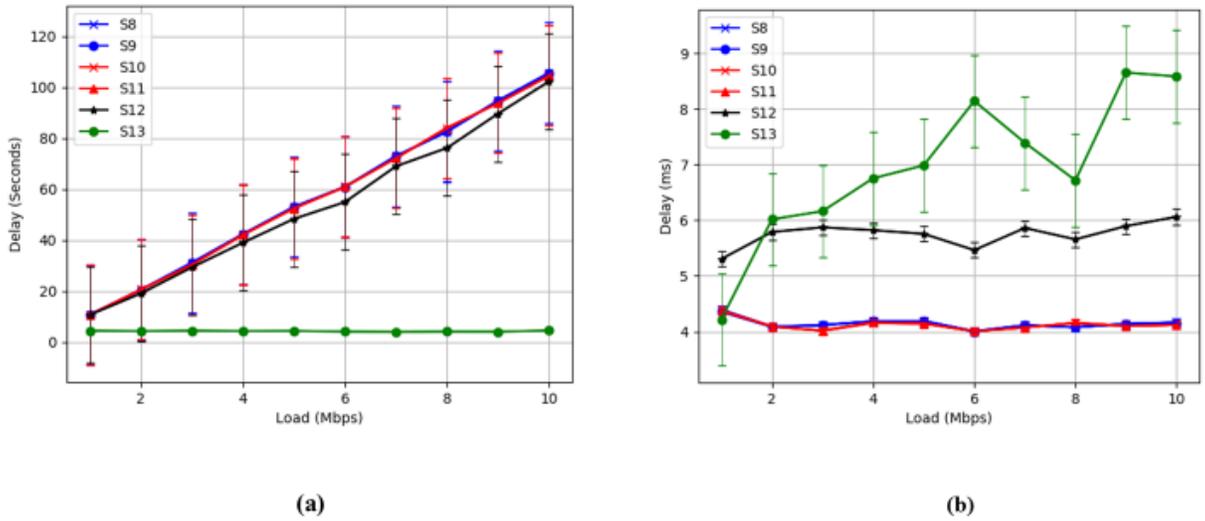


Figure 6.13 Fat-tree decoded traffic delay versus UDP traffic load

6.2.3 CPU Utilization / Computation Power

Figure 6.14 shows the resulted CPU utilization for the encoding nodes in the fat-tree topology. The graph reflects a one sample experiment, when a 5 Mbps UDP traffic was injected for 10 seconds, and the encoding nodes S4, S5 and S6 started decoding the received UDP packets. The CPU consumption behavior of the three encoding processes was like the encoding node in the butterfly scenario, also encoding nodes create threads to generate NC packets and when they finish they release these threads.

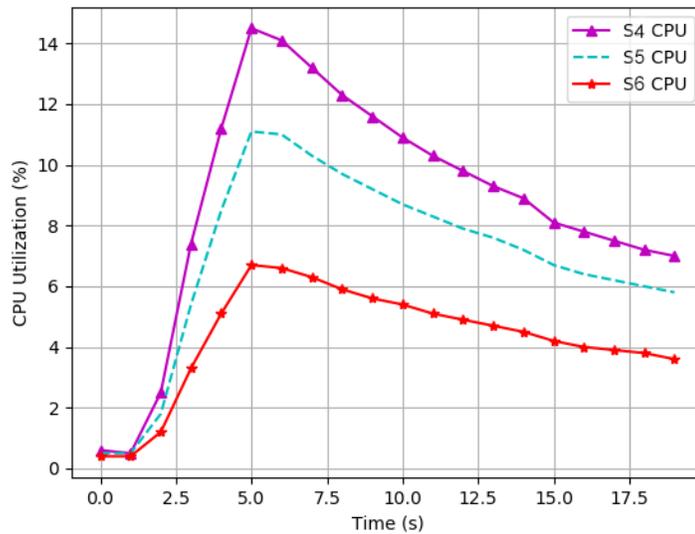


Figure 6.14 Fat-tree CPU utilization for encoding nodes s4, s5 and s6

Figure 6.15 shows the resulted CPU utilization for the decoding nodes of the same sample experiment. The graph shows the CPU consumption behavior of the encoding processes was increasing over time and consumed the all available CPU power of the host virtual machine. As it was noticed the S12 and S13 processes CPU usage was only 17% each and less the CPU usage of the other decoding nodes (S8, S9, S10 , S11) and this limited CPU power affected the decoding process.

The experimental virtual machine has four cores CPU for processing, so the CPU utilization scale is 400%. As we have noticed sum of CPU utilization of encoding and decoding processes are more than 100% scale, because of parallel processing and hyper-threading.

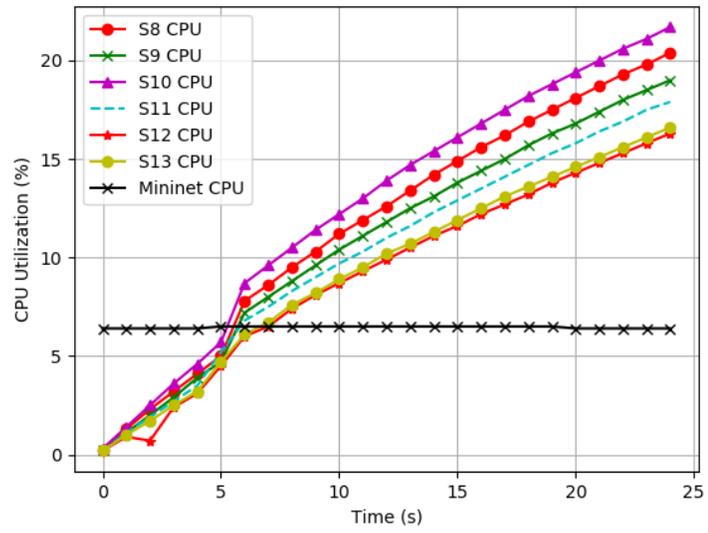


Figure 6.15 Fat-tree CPU Utilization for Decoding nodes S8,S9,S10,S11,S12 and S13

Chapter 7

Summary and Future Work

7.1 Summary

The body of this thesis, consists of five chapters focused on Random Linear Network Coding implementation over Software Defined Networks. In chapter two, various network coding and SDN multicast models were reviewed and studied.

In the third chapter, a new RLNC-SDN architecture was developed to integrate network coding capabilities for both SDN data and control planes. In chapter four, the implementation tools and experimental setup were introduced.

In chapter five, the proposed RLNC-SDN framework was evaluated and analyzed with detailed results in throughput, delay and CPU utilization for butterfly and fat-tree topologies.

7.2 Thesis Accomplishments

This work contributions can be summarized as following:

- Study of network coding theory and gain deep understanding on network coding challenges in multicast networks.
- Study of Software Defined Networking architecture, explore its flexible capabilities and identify the limitations.
- Extending OpenFlow protocol to support RLNC functions and provide SDN controller the ability to adjust the coding parameters.

- A full-fledged RLNC-SDN switch prototype has been developed and implemented in real-like multicast networks.
- A centrality-based routing mechanism has been implemented to optimize the network throughput.
- Performance analysis has been conducted for single-source multicast topologies in terms of throughput, delay and consumed resources.

7.3 Future Work

- Investigate the performance of the developed RLNC-SDN prototype on more complex single or multiple sources multicast networks.
- In this work, we have used a single RLNC codec, some further experiments are required for different RLNC codec types and different coding/decoding parameters e.g. generation size, maximum symbol size...etc.
- Optimize the buffer management module to minimize the stalls in coding and decoding operations.
- Develop an adaptive RLNC routing protocol that would increase the gain of network throughput.

References

- [1] M. Tan, R. W. Yeung and S.-T. Ho, "A Unified Framework for Linear Network," 2008.
- [2] N. C.-Y. L. Y. R. Ahlswede, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204-1216, 2000.
- [3] D. L. Tracey Ho, "What is network coding," in *Network Coding: An Introduction*.
- [4] P. G. F. Jorge Castiñeira Moreira, *Essentials of Error-Control Coding*, Wiley, 2006.
- [5] R. Koetter and M. Médard, "An Algebraic Approach to Network Coding," *ACM Transactions on Networking*, vol. 11, no. 5, pp. 782-795, 2003.
- [6] R. K. M. K. E. Tracey Ho, "The benefits of coding over routing in a randomized setting," in *IEEE International Symposium on Information Theory, 2003. Proceedings*, 2003.
- [7] A. S. Khan and I. Chatzigeorgiou, "Performance Analysis of Random Linear Network Coding in Two-Source Single-Relay Networks," in *IEEE ICC 2015 - Workshop on Cooperative and Cognitive Networks*, 2015.
- [8] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14 - 76, 2014.
- [9] S. Azodolmolky, *software Defined Networking with OpenFlow*, PACKT, 2013.
- [10] "OpenFlow Switch Specification Version 1.5.0," Open Networking Foundation, 2014.
- [11] D. Sheinbein and R. Weber, "Stored Program Controlled Network: 800 Service using SPC network capability," *Bell System Technical Journal*, vol. 61, no. 7, pp. 1737 - 1744, 1982.
- [12] K. Govindarajan, K. C. Meng and H. Ong, "A literature review on Software-Defined Networking (SDN) research topics, challenges and solutions," in *Advanced Computing (ICoAC), 2013 Fifth International Conference*, Chennai, 2013.
- [13] D. Thaler and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection," *ietf*, 2000.
- [14] P. S. C. E. E. J. T. S. Jaggi, "Polynomial time algorithms for multicast network code construction," *IEEE Transactions on Information Theory*, vol. 51, no. 6, pp. 1973-1982, 2005.
- [15] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher and J. Barros, "Network Coding Meets TCP: Theory and Implementation," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490-512, 2011.
- [16] K. S. S. D. Kotani, "A Design and Implementation of OpenFlow Controller Handling IP Multicast with Fast Tree Switching," in *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet (SAINT)*, 2012.
- [17] B. H. Sicheng Liu, "NCoS: A framework for realizing network coding over software-defined network," in *2014 IEEE 39th Conference on Local Computer Networks (LCN)*, 2014.

- [18] A. G. H. F. E. L. David Szabo, "Towards the Tactile Internet: Decreasing Communication Latency with Network Coding and Software Defined Networking," in *European Wireless 2015; 21th European Wireless Conference*, 2015.
- [19] C.-T. L. Chin-Tau, "Network-coding Multicast Networks with QoS Guarantees," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, p. 265–274, 2011.
- [20] H. C.-S. P. Tae-hwa Kim, "Centrality-based network coding node selection mechanism for improving network throughput," in *2014 16th International Conference on Advanced Communication Technology (ICACT)*, 2014.
- [21] R. Y. C. S.-Y.R. Li, "Linear network coding," *IEEE Transactions on Information Theory*, no. 2, pp. 371-381, 2003.
- [22] B. L. G. Ying Zhu, "Multicast with network coding in application-layer overlay networks," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, p. 107–120, 2004.
- [23] B. L. Mea Wang, "R2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming," *IEEE J.Sel. A. Commun.*, vol. 25, no. 9, p. 1655–1666, 2007.
- [24] A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyas, W. Tavernier and S. Sahhaf, "ESCAPE: extensible service chain prototyping environment using mininet, click, NETCONF and POX," in *SIGCOMM '14*, New York, 2014.
- [25] "DPDK," Linux Foundation Project, [Online]. Available: <http://dpdk.org/>.
- [26] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Section 24.3: Dijkstra's algorithm," in *Introduction to Algorithms*, McGraw–Hill, 2001, p. 595–601.
- [27] "iPerf -Test tool for TCP, UDP and SCTP," [Online]. Available: <https://iperf.fr/>.
- [28] "Kodo-RLNC documentation," [Online]. Available: <http://docs.steinwurf.com>.
- [29] "Mininet SDN Emulator," [Online]. Available: <http://mininet.org/>.
- [30] "Wireshark network protocol analyzer," [Online]. Available: <https://www.wireshark.org/>.
- [31] "RYU SDN Framework," [Online]. Available: <https://osrg.github.io/ryu/>.
- [32] "NetworkX, Network functions python package," [Online]. Available: <http://networkx.github.io/>.
- [33] "OpenvSwitch," [Online]. Available: <http://openvswitch.org/>.
- [34] "Kodo RLNC Libraries," [Online]. Available: <http://steinwurf.com/>.
- [35] E. Jo, D. Pan and J. Liu, "A simulation and emulation study of SDN-based multipath routing for fat-tree data center networks," in *Simulation Conference (WSC), 2014 Winter*, Savannah, GA, USA, 2014.

Appendices

A. OpenVswitch RLNC modifications

- `/include/odp-netlink.h`

```
enum ovs_action_attr {
OVS_ACTION_ATTR_UNSPEC,
OVS_ACTION_ATTR_OUTPUT,          /* u32 port number. */
OVS_ACTION_ATTR_USERSPACE,      /* Nested OVS_USERSPACE_ATTR*. */
OVS_ACTION_ATTR_SET,            /* One nested OVS_KEY_ATTR*. */
OVS_ACTION_ATTR_PUSH_VLAN,      /* struct ovs_action_push_vlan. */
OVS_ACTION_ATTR_POP_VLAN,       /* No argument. */
OVS_ACTION_ATTR_SAMPLE,         /* Nested OVS_SAMPLE_ATTR*. */
OVS_ACTION_ATTR_RECIRC,         /* u32 recirc_id. */
OVS_ACTION_ATTR_HASH,           /* struct ovs_action_hash. */
OVS_ACTION_ATTR_PUSH_MPLS,      /* struct ovs_action_push_mpls. */
OVS_ACTION_ATTR_POP_MPLS,       /* ovs_be16 ethertype. */
OVS_ACTION_ATTR_SET_MASKED,     /* One nested OVS_KEY_ATTR*
including
                                * data immediately followed by a
mask.
                                * The data must be zero for the
unmasked
                                * bits. */
OVS_ACTION_ATTR_NC_ENCODE,      /* struct ovs_action_nc_encode*/
OVS_ACTION_ATTR_NC_DECODE,     /* struct ovs_action_nc_decode*/
}
```

```

enum ovs_key_attr {
    OVS_KEY_ATTR_UNSPEC,
    OVS_KEY_ATTR_ENCAP, /* Nested set of encapsulated
attributes. */
    OVS_KEY_ATTR_PRIORITY, /* u32 skb->priority */
    OVS_KEY_ATTR_IN_PORT, /* u32 OVS dp port number */
    OVS_KEY_ATTR_ETHERNET, /* struct ovs_key_ethernet */
    OVS_KEY_ATTR_VLAN, /* be16 VLAN TCI */
    OVS_KEY_ATTR_ETHERTYPE, /* be16 Ethernet type */
    OVS_KEY_ATTR_IPV4, /* struct ovs_key_ipv4 */
    OVS_KEY_ATTR_IPV6, /* struct ovs_key_ipv6 */
    OVS_KEY_ATTR_TCP, /* struct ovs_key_tcp */
    OVS_KEY_ATTR_UDP, /* struct ovs_key_udp */
    .....
    .....
    OVS_KEY_ATTR_NC_ENCODE, /* struct ovs_key_nc_encode */
    OVS_KEY_ATTR_NC_ENCODE, /* struct ovs_key_nc_decode */
}

```

```
struct ovs_key_ethernet {
    uint8_t    eth_src[ETH_ADDR_LEN];
    uint8_t    eth_dst[ETH_ADDR_LEN];
};
....
/* struct of encode key attributes */
struct ovs_key_nc_encode {
    ovs_be16 buffer_id;
    ovs_be16 gen_size;
    ovs_be16 output_num;
    ovs_be16 input_num;
    ovs_be16 out_port;
};
/* struct of decode key attributes */
struct ovs_key_nc_decode {
    ovs_be16 buffer_id;
    ovs_be16 gen_size;
    ovs_be16 output_num;
    ovs_be16 input_num;
    ovs_be16 out_port;
};
```

```

struct ovs_action_push_mpls {
    ovs_be32 mpls_lse;
    ovs_be16 mpls_ethertype; /* Either %ETH_P_MPLS_UC or
%ETH_P_MPLS_MC */
};
....
/* struct ovs_action_nc_encode - for networkcoding action */
struct ovs_action_nc_encode {
    uint16_t buffer_id;
    uint16_t generation_size;
    uint16_t input_num;
    uint16_t output_num;
    uint16_t outports;
};
/* struct ovs_action_nc_decode - for decode action */
struct ovs_action_nc_encode {
    uint16_t buffer_id;
    uint16_t generation_size;
    uint16_t input_num;
    uint16_t output_num;
    uint16_t outports;
};

```

- /datapath/actions.c

```

static int nc_encode(struct datapath *dp, struct sw_flow_key
*key, const struct nlattr *attr) {
pr_warn("%s: nc_encode is reached here\n", ovs_dp_name(dp));
const struct nlattr *acts_list = NULL;
    const struct nlattr *a;
    int rem;
    for (a = nla_data(attr), rem = nla_len(attr); rem > 0;
        a = nla_next(a, &rem)) {
        switch (nla_type(a)) {
        case OVS_ACTION_ATTR_NC_ENCODE:
            acts_list = a;
            break;
        }
    }
    rem = nla_len(acts_list);
    a = nla_data(acts_list);
pr_warn("action is triggered");
    //if (unlikely(!rem))
        return 0;
}

```

```

static int nc_decode(struct datapath *dp, struct sw_flow_key
*key, const struct nlattrib *attr) {
pr_warn("%s: nc_decode is reached here\n", ovs_dp_name(dp));
const struct nlattrib *acts_list = NULL;
    const struct nlattrib *a;
    int rem;
    for (a = nla_data(attr), rem = nla_len(attr); rem > 0;
        a = nla_next(a, &rem)) {
        switch (nla_type(a)) {
        case OVS_ACTION_ATTR_NC_DECODE:
            acts_list = a;
            break;
        }
    }
    rem = nla_len(acts_list);
    a = nla_data(acts_list);
pr_warn("action is triggered");
    //if (unlikely(!rem))
        return 0;
}

```

```

static int
do_execute_actions(struct datapath *dp, struct sk_buff *skb, struct
sw_flow_key *key, const struct nlattr *attr, int len)
{
    int prev_port = -1;
    const struct nlattr *a;
    int rem;

    for (a = attr, rem = len; rem > 0;
         a = nla_next(a, &rem)) {
        int err = 0;

        if (unlikely(prev_port != -1)) {
            struct sk_buff *out_skb = skb_clone(skb, GFP_ATOMIC);
            if (out_skb)
                do_output(dp, out_skb, prev_port);

            prev_port = -1;
        }

        switch (nla_type(a)) {
        case OVS_ACTION_ATTR_OUTPUT:
            prev_port = nla_get_u32(a);
            break;
        ....
        case OVS_ACTION_ATTR_NC_ENCODE:
            err = nc_encode(dp, key, a);
            break;
        case OVS_ACTION_ATTR_NC_DECODE:
            err = nc_decode(dp, key, a);
            break;
        }
    }
}

```

- /datapath/Flow_netlink.c

```

static int __ovs_nla_copy_actions(const struct nlattrib *attr, const
struct sw_flow_key *key, int depth, struct sw_flow_actions **sfa,
__be16 eth_type, __be16 vlan_tci, bool log)
{
    const struct nlattrib *a;
    int rem, err;

    if (depth >= SAMPLE_ACTION_DEPTH)
        return -E_OVERFLOW;

    nla_for_each_nested(a, attr, rem) {
/* Expected argument lengths, (u32)-1 for variable length. */
static const u32 action_lens[OVS_ACTION_ATTR_MAX + 1] = {
    [OVS_ACTION_ATTR_OUTPUT] = sizeof(u32),
    ...
    [OVS_ACTION_ATTR_NC_ENCODE] = 0,
    [OVS_ACTION_ATTR_NC_DECODE] = 0,
    ...
};
const struct ovs_action_push_vlan *vlan;
int type = nla_type(a);
bool skip_copy;
    printk("%d", type);
    if (type > OVS_ACTION_ATTR_MAX ||
        (action_lens[type] != nla_len(a) &&
         action_lens[type] != (u32)-1))
        return -EINVAL;

    skip_copy = false;
    switch (type) {
case OVS_ACTION_ATTR_UNSPEC:
        return -EINVAL;

case OVS_ACTION_ATTR_USERSPACE:
        err = validate_userspace(a);
        if (err)
            return err;
        break;

case OVS_ACTION_ATTR_OUTPUT:
        if (nla_get_u32(a) >= DP_MAX_PORTS)
            return -EINVAL;

        break;
    ...
    ...
    case OVS_ACTION_ATTR_NC_ENCODE:
        err=0;

```

- /ofproto/ofproto-dpif-xlate.c

```

static void
recirc_unroll_actions(const struct ofpact *ofpacts, size_t
ofpacts_len, struct xlate_ctx *ctx)
{
    const struct ofpact *a;

    OFPACT_FOR_EACH (a, ofpacts, ofpacts_len) {
        switch (a->type) {
            /* May generate PACKET INs. */
            case OFPACT_OUTPUT_REG:
            case OFPACT_GROUP:
            case OFPACT_OUTPUT:
            case OFPACT_CONTROLLER:
            case OFPACT_NC_ENCODE:
            case OFPACT_NC_DECODE:
                break;

            /* These need not be copied for restoration. */
            case OFPACT_NOTE:
            case OFPACT_CONJUNCTION:
                continue;
        }
        /* Copy the action over. */
        ofpbuf_put(&ctx->action_set, a, OFPACT_ALIGN(a->len));
    }
}

```

```

/* network coding compose functions */
static void
compose_nc_encode_action(struct xlate_ctx *ctx){
    struct dp_packet *packet;
    xlate_report(ctx,"encoding is started !");
    nl_msg_put_u32(ctx->xout-
>odp_actions,OVS_ACTION_ATTR_NC_ENCODE,0);
    if(ctx->xin->packet != NULL ){
        packet = dp_packet_clone(ctx->xin->packet);
        nc_encode(packet);
        ctx->xout->slow |= commit_odp_actions(&ctx->xin->flow,
&ctx->base_flow,ctx->xout->odp_actions,&ctx->xout->wc,ctx-
>xbridge->support.masked_set_action);
    }

compose_nc_decode_action(struct xlate_ctx *ctx){
    struct dp_packet *packet;
    xlate_report(ctx,"encoding is started !");
    nl_msg_put_u32(ctx->xout-
>odp_actions,OVS_ACTION_ATTR_NC_DECODE,0);
    if(ctx->xin->packet != NULL ){
        packet = dp_packet_clone(ctx->xin->packet);
        nc_decode(packet);
        ctx->xout->slow |= commit_odp_actions(&ctx->xin->flow,
&ctx->base_flow,ctx->xout->odp_actions,&ctx->xout->wc,ctx-
>xbridge->support.masked_set_action);
    }
}

```

```

static void
do_xlate_actions(const struct ofpact *ofpacts, size_t
ofpacts_len, struct xlate_ctx *ctx)
{
    struct flow_wildcards *wc = &ctx->xout->wc;
    struct flow *flow = &ctx->xin->flow;
    const struct ofpact *a;

    if (ovs_native_tunneling_is_on(ctx->xbridge->ofproto)) {
        tnl_arp_snoop(flow, wc, ctx->xbridge->name);
    }
    /* dl_type already in the mask, not set below. */

    OFPACT_FOR_EACH (a, ofpacts, ofpacts_len) {
        struct ofpact_controller *controller;
        const struct ofpact_metadata *metadata;
        const struct ofpact_set_field *set_field;
        const struct mf_field *mf;

        if (ctx->exit) {
            /* Check if need to store the remaining actions for later
             * execution. */
            if (exit_recirculates(ctx)) {
                recirc_unroll_actions(a, OFPACT_ALIGN(ofpacts_len -
((uint8_t *)a - (uint8_t *)ofpacts)), ctx);
            }
            break;
        }

        switch (a->type) {
        case OFPACT_OUTPUT:
            xlate_output_action(ctx, ofpact_get_OUTPUT(a)->port,
ofpact_get_OUTPUT(a)->max_len, true);
            break;
        ...
        ...
        case OFPACT_NC_ENCODE:
            compose_nc_encode_action(ctx);
            break;
        }
        case OFPACT_NC_DECODE:
            compose_nc_decode_action(ctx);
            break;
        }

        /* Check if need to store this and the remaining actions for
        later * execution. */
        if (ctx->exit && ctx_first_recirculation_action(ctx)) {
            recirc_unroll_actions(a, OFPACT_ALIGN(ofpacts_len -
((uint8_t *)a - (uint8_t *)ofpacts)), ctx);
            break;
        }
    }
}

```

- `/lib/dpif-netdev.c`

```

static int
nc_encode_action (struct dp_packet **packets, int cnt)
{
uint8_t max_symbols = 20;
uint32_t max_symbol_size = 512;
struct dp_packet *packet_copy;
int32_t code_type = kodo_full_vector;
int32_t finite_field = kodo_binary;
kodo_factory_t encoder_factory =
    kodo_new_encoder_factory(code_type, finite_field,
                            max_symbols, max_symbol_size);
kodo_coder_t encoder = kodo_factory_new_encoder(encoder_factory);
uint32_t block_size = kodo_block_size(encoder);
uint8_t* data_in = (uint8_t*) malloc(block_size);
uint32_t payload_size = kodo_payload_size(encoder);
uint8_t* payload = (uint8_t*) malloc(payload_size);
int i = 0;
for(; i < cnt; ++i){
    packet_copy =
dp_packet_clone_data(packets[i], sizeof(packets[i]));
    if(sizeof(packet_copy) != 0){
        data_in[i] =(uint8_t)
atoi(dp_packet_to_string(packet_copy, sizeof(packet_copy)));
    }else{
        return -EINVAL;
    }
}
kodo_set_const_symbols(encoder, data_in, block_size);
if (kodo_is_systematic_on(encoder)){
    //printf("Turning systematic OFF\n");
    kodo_set_systematic_off(encoder);
}
uint32_t bytes_used = 0;
bytes_used = kodo_write_payload(encoder, payload);
//printf("coded payload:%d", bytes_used);
//coded_packet = dp_packet_clone_data(payload, bytes_used);
//debug_packet = dp_packet_to_string(coded_packet, 1024);
struct dp_packet *b = dp_packet_new(sizeof(payload));
dp_packet_put(b, payload, sizeof(payload));
free(data_in);
free(payload);
kodo_delete_coder(encoder);
kodo_delete_factory(encoder_factory);
return 0;
}

```

```

static int
nc_decode_action (struct dp_packet **packets, int cnt)
{
uint8_t max_symbols = 20;
uint32_t max_symbol_size = 512;
struct dp_packet *packet_copy;
int32_t code_type = kodo_full_vector;
int32_t finite_field = kodo_binary;
kodo_factory_t decoder_factory =
    kodo_new_decoder_factory(code_type, finite_field,
                            max_symbols, max_symbol_size);
kodo_coder_t decoder = kodo_factory_new_decoder(decoder_factory);
uint32_t block_size = kodo_block_size(decoder);
uint8_t* data_in = (uint8_t*) malloc(block_size);
uint32_t payload_size = kodo_payload_size(decoder);
uint8_t* payload = (uint8_t*) malloc(payload_size);
int i = 0;
for(; i < cnt; ++i){
    packet_copy =
dp_packet_clone_data(packets[i], sizeof(packets[i]));
    if(sizeof(packet_copy) != 0){
        data_in[i] = (uint8_t)
atoi(dp_packet_to_string(packet_copy, sizeof(packet_copy)));
    }else{
        return -EINVAL;
    }
}
uint32_t bytes_used = 0;
bytes_used = kodo_read_payload(decoder, payload);
//printf("decoded payload:%d", bytes_used);
//coded_packet = dp_packet_clone_data(payload, bytes_used);
//debug_packet = dp_packet_to_string(decoded_packet, 1024);
struct dp_packet *b = dp_packet_new(sizeof(payload));
dp_packet_put(b, payload, sizeof(payload));
free(data_in);
free(payload);
kodo_delete_coder(decoder);
kodo_delete_factory(decoder_factory);
return 0;
}

```

```

static void
dp_execute_cb(void *aux_, struct dp_packet **packets, int cnt, const
struct nlattr *a, bool may_steal)
{
struct dp_netdev_execute_aux *aux = aux_;
uint32_t *depth = recirc_depth_get();
struct dp_netdev_pmd_thread *pmd = aux->pmd;
struct dp_netdev *dp = pmd->dp;
int type = nl_attr_type(a);
struct dp_netdev_port *p;
int i;
switch ((enum ovs_action_attr)type) {
case OVS_ACTION_ATTR_OUTPUT:
p = dp_netdev_lookup_port(dp,
u32_to_odp(nl_attr_get_u32(a)));
if (OVS_LIKELY(p)) {
netdev_send(p->netdev, pmd->tx_qid, packets,
cnt, may_steal);
return;
}
break;
...
case OVS_ACTION_ATTR_NC_ENCODE:
if (*depth < MAX_RECIRC_DEPTH) {
struct dp_packet *tnl_pkt[NETDEV_MAX_BURST];
int err;

if (!may_steal) {
dp_netdev_clone_pkt_batch(tnl_pkt, packets, cnt);
packets = tnl_pkt;}
err = nc_encode_action(packets, cnt);
if (!err) {
(*depth)++;
dp_netdev_input(pmd, packets, cnt);
(*depth)--;
} else {
dp_netdev_drop_packets(tnl_pkt, cnt, !may_steal);
}
return;
}
break;
}
}

```

- `/lib/ofp-actions.c`

```
enum ofp_raw_action_type {
/* ## ----- ## */
/* ## Standard actions. ## */
/* ## ----- ## */
    /* OF1.0(0): struct ofp10_action_output. */
    OFPAT_RAW10_OUTPUT,
...
...
    /* NC1.1+(1): struct nc_action_encode. */
    OFPAT_RAW_NC_ENCODE
    /* NC1.1+(2): struct nc_action_decode. */
    OFPAT_RAW_NC_DECODE
}
```

- /lib/odp-util.c

```

static int
odp_action_len(uint16_t type)
{
    if (type > OVS_ACTION_ATTR_MAX) {
        return -1;
    }

    switch ((enum ovs_action_attr) type) {
    case OVS_ACTION_ATTR_OUTPUT: return sizeof(uint32_t);
    ...
    ...
    case OVS_ACTION_ATTR_NC_ENCODE: return ATTR_LEN_VARIABLE;
    case OVS_ACTION_ATTR_NC_DECODE: return ATTR_LEN_VARIABLE;
    case OVS_ACTION_ATTR_UNSPEC:
    case __OVS_ACTION_ATTR_MAX:
        return ATTR_LEN_INVALID;
    }
}

static void
format_odp_nc_encode_action(struct ds *ds, const struct nlattr *attr)
{
    struct ovs_action_nc_encode *data;
    data = (struct ovs_action_nc_encode *) nl_attr_get(attr);
    ds_put_cstr(ds, "nc_encode(");
    ds_put_format(ds, "buffer_id=0x%"PRIu16, data->buffer_id);
    ds_put_format(ds, ", gen_size=%"PRIu16, data->generation_size);
    ds_put_format(ds, ", output_num=%"PRIu16, data->output_num);
    ds_put_format(ds, ", input_num=%"PRIu16, data->input_num);
    ds_put_format(ds, ", out_port=0x%"PRIu16, data->outports);
    ds_put_format(ds, ")");
}

static void
format_odp_nc_decode_action(struct ds *ds, const struct nlattr *attr)
{
    struct ovs_action_nc_decode *data;
    data = (struct ovs_action_nc_decode *) nl_attr_get(attr);
    ds_put_cstr(ds, "nc_decode(");
    ds_put_format(ds, "buffer_id=0x%"PRIu16, data->buffer_id);
    ds_put_format(ds, ", gen_size=%"PRIu16, data->generation_size);
    ds_put_format(ds, ", output_num=%"PRIu16, data->output_num);
    ds_put_format(ds, ", input_num=%"PRIu16, data->input_num);
    ds_put_format(ds, ", out_port=0x%"PRIu16, data->outports);
    ds_put_format(ds, ")");
}

```

```

Static void
format_odp_action(struct ds *ds, const struct nlattr *a)
{
    int expected_len;
    enum ovs_action_attr type = nl_attr_type(a);
    const struct ovs_action_push_vlan *vlan;
    size_t size;

    expected_len = odp_action_len(nl_attr_type(a));
    if (expected_len != ATTR_LEN_VARIABLE &&
        nl_attr_get_size(a) != expected_len) {
        ds_put_format(ds, "bad length %"PRIuSIZE", expected %d
for: ",
                    nl_attr_get_size(a), expected_len);
        format_generic_odp_action(ds, a);
        return;
    }

    switch (type) {
    case OVS_ACTION_ATTR_OUTPUT:
        ds_put_format(ds, "%"PRIu32, nl_attr_get_u32(a));
        break;
        ...
        ...
    case OVS_ACTION_ATTR_NC_ENCODE:{
        format_odp_nc_encode_action(ds,a);
        break;
    case OVS_ACTION_ATTR_NC_DECODE:{
        format_odp_nc_decode_action(ds,a);
        break;
default:
        format_generic_odp_action(ds, a);
        break;
    }
}
}

```

```

static const char *
ovs_key_attr_to_string(enum ovs_key_attr attr, char *namebuf,
size_t bufsize)
{
    switch (attr) {
        case OVS_KEY_ATTR_UNSPEC: return "unspec";
        case OVS_KEY_ATTR_IN_PORT: return "in_port";
        ...
        ...
        case OVS_KEY_ATTR_NC_ENCODE: return "nc_encode";
        case OVS_KEY_ATTR_NC_DECODE: return "nc_decode";
        case __OVS_KEY_ATTR_MAX:
        default:
            sprintf(namebuf, bufsize, "key%u", (unsigned int) attr);
            return namebuf;
    }
}

static const struct attr_len_tbl
ovs_flow_key_attr_lens[OVS_KEY_ATTR_MAX + 1] = {
[OVS_KEY_ATTR_IN_PORT] = { .len = 4 },
    [OVS_KEY_ATTR_ETHERNET] = { .len = sizeof(struct
ovs_key_ethernet) },
    [OVS_KEY_ATTR_VLAN] = { .len = 2 },
    [OVS_KEY_ATTR_ETHERTYPE] = { .len = 2 },
    [OVS_KEY_ATTR_MPLS] = { .len = ATTR_LEN_VARIABLE },
    [OVS_KEY_ATTR_IPV4] = { .len = sizeof(struct
ovs_key_ipv4) },
    [OVS_KEY_ATTR_IPV6] = { .len = sizeof(struct
ovs_key_ipv6) },
    ...
    ...
[OVS_KEY_ATTR_NC_ENCODE] = { .len = ATTR_LEN_VARIABLE },
[OVS_KEY_ATTR_NC_DECODE] = { .len = ATTR_LEN_VARIABLE },
}

```

- **/lib/odp-execute.c**

```
#include Packet.h
static void
odp_execute_nc_encode(struct dp_packet *packet)
{
    struct eth_header *eh = dp_packet_l2(packet);
    if(eh) {
        printf("network coding packet generation");
        nc_encode(packet);
    }
}
static void
odp_execute_nc_decode(struct dp_packet *packet)
{
    struct eth_header *eh = dp_packet_l2(packet);
    if(eh) {
        printf("decoding nc packet");
        nc_decode(packet);
    }
}
```

- **/lib/packet.c**

```
void nc_encode(struct dp_packet *b) {
    if(sizeof(b) != 0)
    {
        dp_packet_clear(b);
    }
}
void nc_decode(struct dp_packet *b) {
    if(sizeof(b) != 0)
    {
        dp_packet_clear(b);
    }
}
```

- **/lib/packet.h**

```
void nc_encode(struct dp_packet *);
```

B. Mininet Topology scripts

• **butter-fly-topology.py**

```
#!/usr/bin/python

"""
This Mininet butterfly network coding topology.
"""

from mininet.net import Mininet
from mininet.node import Controller
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.topo import Topo
from mininet.link import TCLink
from mininet.log import setLogLevel, info
from mininet.node import OVSSwitch

class ButterflyTopo(Topo,OVSSwitch):
    def __init__( self, **params ):
        # Initialize topology
        Topo.__init__( self, **params )
        h1 = self.addHost('h1',ip="10.0.0.1")
        h2 = self.addHost('h2',ip="10.0.0.2")
        sw1 = self.addSwitch('s1',cls=OVSSwitch,protocols='OpenFlow13')
        sw2 = self.addSwitch('s2',cls=OVSSwitch,protocols='OpenFlow13')
        sw3 = self.addSwitch('s3',cls=OVSSwitch,protocols='OpenFlow13')
        sw4 = self.addSwitch('s4',cls=OVSSwitch,protocols='OpenFlow13')
        sw5 = self.addSwitch('s5',cls=OVSSwitch,protocols='OpenFlow13')
        sw6 = self.addSwitch('s6',cls=OVSSwitch,protocols='OpenFlow13')
        sw7 = self.addSwitch('s7',cls=OVSSwitch,protocols='OpenFlow13')
        sw8 = self.addSwitch('s8',cls=OVSSwitch,protocols='OpenFlow13')
        self.addLink(h1,sw1)
        self.addLink(sw1,sw2,bw=10)
        self.addLink(sw1,sw3,bw=10)
        self.addLink(sw2,sw4,bw=10)
        self.addLink(sw2,sw5,bw=10)
        self.addLink(sw3,sw5,bw=10)
        self.addLink(sw3,sw7,bw=10)
        self.addLink(sw4,sw8,bw=10)
        self.addLink(sw5,sw6,bw=10)
        self.addLink(sw6,sw4,bw=10)
        self.addLink(sw6,sw7,bw=10)
        self.addLink(sw7,sw8,bw=10)
        self.addLink(sw8,h2)

def ncTopology ():
    topo = ButterflyTopo()
    net = Mininet(topo=topo,
controller=RemoteController,switch=OVSSwitch,link=TCLink)
    info( '*** Adding controller\n' )
    #net.addController('c0')
    info( '*** Starting network\n' )

    net.start()

    info( '*** Running CLI\n' )
    CLI( net )

    info( '*** Stopping network' )
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'debug' )
```

- **multihop-topology.py**

```
#!/usr/bin/python

"""
This Mininet butterfly network coding example.
"""
from mininet.net import Mininet
from mininet.node import Controller
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.topo import Topo
from mininet.link import TCLink
from mininet.log import setLogLevel, info
from mininet.node import OVSSwitch

class MultiHopTopo(Topo):
    def __init__( self, **params ):
        # Initialize topology
        Topo.__init__( self, **params )
        h1 = self.addHost('h1',ip="10.0.0.1")
        h2 = self.addHost('h2',ip="10.0.0.2")
        sw1 = self.addSwitch('s1')
        sw2 = self.addSwitch('s2')
        self.addLink(h1,sw1)
        self.addLink(sw1,sw2,bw=10)
        self.addLink(sw2,h2,bw=10)

def multihopTopology ():
    topo = MultiHopTopo()
    net = Mininet(topo=topo,
    controller=RemoteController,switch=OVSSwitch,link=TCLink)
    info( '*** Adding controller\n' )
    #net.addController('c0')
    info( '*** Starting network\n' )

    net.start()

    info( '*** Running CLI\n' )
    CLI( net )

    info( '*** Stopping network' )
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    multihopTopology()
```

- ***Fat-tree-topology.py***

```

#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.node import CPULimitedHost, Host, Node
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
from mininet.topo import Topo
from mininet.node import OVSSwitch

class FattreeTopo(Topo,OVSSwitch):
    def __init__(self, **params):
        # Initialize topology
        Topo.__init__(self, **params)

        s1 = self.addSwitch('s1',cls=OVSSwitch,protocols='OpenFlow13')
        s2 = self.addSwitch('s2',cls=OVSSwitch,protocols='OpenFlow13')
        s3 = self.addSwitch('s3',cls=OVSSwitch,protocols='OpenFlow13')
        s4 = self.addSwitch('s4',cls=OVSSwitch,protocols='OpenFlow13')
        s5 = self.addSwitch('s5',cls=OVSSwitch,protocols='OpenFlow13')
        s6 = self.addSwitch('s6',cls=OVSSwitch,protocols='OpenFlow13')
        s7 = self.addSwitch('s7',cls=OVSSwitch,protocols='OpenFlow13')
        s8 = self.addSwitch('s8',cls=OVSSwitch,protocols='OpenFlow13')
        s9 = self.addSwitch('s9',cls=OVSSwitch,protocols='OpenFlow13')
        s10 =
self.addSwitch('s10',cls=OVSSwitch,protocols='OpenFlow13')
        s11 =
self.addSwitch('s11',cls=OVSSwitch,protocols='OpenFlow13')
        s12 =
self.addSwitch('s12',cls=OVSSwitch,protocols='OpenFlow13')
        s13 =
self.addSwitch('s13',cls=OVSSwitch,protocols='OpenFlow13')

        h3 = self.addHost('h3',ip='10.0.0.6',)
        h4 = self.addHost('h4',ip='10.0.0.5',)
        h5 = self.addHost('h5',ip='10.0.0.4',)
        h6 = self.addHost('h6',ip='10.0.0.3',)
        h2 = self.addHost('h2',ip='10.0.0.7',)
        h7 = self.addHost('h7',ip='10.0.0.2',)
        h1 = self.addHost('h1',ip='10.0.0.1',mac='9a:44:54:5c:bb:7d')

        self.addLink(h1, s1)
        self.addLink(s1, s2,bw=10)
        self.addLink(s1, s3,bw=10)
        self.addLink(s1, s4,bw=10)
        self.addLink(s2, s5,bw=10)
        self.addLink(s3, s5,bw=10)
        self.addLink(s3, s6,bw=10)
        self.addLink(s2, s6,bw=10)
        self.addLink(s4, s6,bw=10)
        self.addLink(s4, s7,bw=10)
        self.addLink(s3, s7,bw=10)

```

```

        self.addLink(s2, s7,bw=10)
        self.addLink(s4, s5,bw=10)
        self.addLink(s5, s8,bw=10)
        self.addLink(s5, s9,bw=10)
        self.addLink(s6, s10,bw=10)
        self.addLink(s6, s11,bw=10)
        self.addLink(s7, s12,bw=10)
        self.addLink(s7, s13,bw=10)
        self.addLink(s13, h2)
        self.addLink(s12, h3)
        self.addLink(s11, h4)
        self.addLink(s10, h5)
        self.addLink(s9, h6)
        self.addLink(s8, h7)
def myNetwork():
    topo = FattreeTopo()
    net = Mininet(
topo=topo,controller=RemoteController,switch=OVSSwitch,link=TCLink,ipBase='10.0.0.0/8')
    info( '*** Starting network\n')
    net.start()
    info( '*** Starting controllers\n')
    info( '*** Starting switches\n')
    info('*** Running CLI\n')
    CLI(net)
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()

```

C. Ryu Controller Applications

- **Ryu/app/butter-fly.py**

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3

class NcFlows (app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(NcFlows, self).__init__(*args, **kwargs)
        # initialize mac address table.
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        dpid = ev.msg.datapath_id
        if dpid == 5:
            # match = parser.OFPMatch(eth_type=34935)
            match = parser.OFPMatch(in_port=1)
            # @type ofproto
            actions =
[parser.OFPActionOutput(ofproto.OFPIT_CLEAR_ACTIONS)]
            self.add_flow(datapath, 0, match, actions)
            match2 = parser.OFPMatch(in_port=2)
            actions2 = [parser.OFPActionOutput(3)]
            self.add_flow(datapath, 0, match2, actions2)
        if dpid in [1,2,3]:
            match = parser.OFPMatch(in_port=1)
            actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            self.add_flow(datapath, 0, match, actions)
        if dpid == 7:
            match = parser.OFPMatch(in_port=2)
            actions = [parser.OFPActionOutput(3)]
            self.add_flow(datapath, 0, match, actions)
        if dpid == 4:
            match = parser.OFPMatch(in_port=3)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath, 0, match, actions)
        if dpid == 6:
            match = parser.OFPMatch(in_port=1)
            actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            self.add_flow(datapath, 0, match, actions)
        if dpid in [1,2,4]:
            match = parser.OFPMatch(in_port=2)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 1, match, actions)
        if dpid == 8:
            match = parser.OFPMatch(in_port=1)
            actions = [parser.OFPActionOutput(3)]
            self.add_flow(datapath, 0, match,actions)
            match = parser.OFPMatch(in_port=2)
            actions = [parser.OFPActionOutput(3)]

```

- **Ryu/app/fat-tree-controller-app.py**

```

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3

class FatNCFlows (app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(FatNCFlows, self).__init__(*args, **kwargs)
        # initialize mac address table.

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        dpid = ev.msg.datapath_id

        if dpid in [1 ,2 ,3 ,4]:
            match = parser.OFPMatch(in_port=1)
            actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]
            self.add_flow(datapath, 0, match, actions)

        if dpid in [8,9,10,11,12,13]:
            match = parser.OFPMatch(in_port=1)
            actions = [parser.OFPActionOutput(2)]
            self.add_flow(datapath, 0, match, actions)
        if dpid in [8,9,10,11,12,13]:
            match = parser.OFPMatch(in_port=2,eth_type=2054)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=2, eth_type=2048)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
        if dpid in [5 ,6 ,7]:
            match = parser.OFPMatch(in_port=4,eth_type=2048)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=5,eth_type=2048)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=4, eth_type=2054)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=5, eth_type=2054)
            actions = [parser.OFPActionOutput(1)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=1)
            actions =
[parser.OFPActionOutput(4),parser.OFPActionOutput(5)]
            self.add_flow(datapath, 0, match, actions)
            match = parser.OFPMatch(in_port=2)

```

```

        actions =
[parser.OFPActionOutput(4),parser.OFPActionOutput(5)]
        self.add_flow(datapath, 0, match, actions)
        match = parser.OFPMatch(in_port=3)
        actions =
[parser.OFPActionOutput(4),parser.OFPActionOutput(5)]
        self.add_flow(datapath, 0, match, actions)
    if dpid in [ 3, 4]:
        match = parser.OFPMatch(in_port=3,eth_type=2054)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, 0, match, actions)
    if dpid == 2:
        match = parser.OFPMatch(in_port=2,eth_type=2054)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, 0, match, actions)
    if dpid == 1:
        match = parser.OFPMatch(in_port=2)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, 0, match, actions)
        match = parser.OFPMatch(in_port=3)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, 0, match, actions)
        match = parser.OFPMatch(in_port=4)
        actions = [parser.OFPActionOutput(1)]
        self.add_flow(datapath, 0, match, actions)

def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    # construct flow_mod message and send it.
    inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                             actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                             match=match, instructions=inst)
    datapath.send_msg(mod)

```

Vitae

- Name : Ahmed Ali Mohammed Hassan
- Nationality : Sudanese
- Date of Birth :10/7/1987
- Email : phpmax@gmail.com
- Address : King Saud St.,Al-Halabi Building, Taif, Saudi Arabia.
- Academic Background : M.Sc. in Computer Engineering, Majoring in
Computer Networks Engineering, KFUPM 2018.
B.Sc. in Electronics Engineering, Majoring in
Telecommunication Engineering, Sudan University of
Science and Technology (2010).
- Work Experience :
- Research Assistant - King Fahd University of
Petroleum and Minerals (2014-2016),
Dhahran,Saudi Arabia.
 - IoT Project Manager – NEARMOTION Co. Ltd.
(2016-2018),Khobar, Saudi Arabia.
 - Product Development Manager – Machinestalk Co.
Ltd. (2018 – to present), Riaydh, Saudi Arabia.

Publications and Patents

Ashraf H. Mahmoud, **Ahmed Hassan**, Marwan Abu-Amara and Tarig Sheltami
“Realization of Random Linear Network Coding over Software Defined Networks”, In
5th IEEE International Conference on Network Softwarization. (Submitted)