An OpenFlow-based Software-Defined Networking Implementation over Named Data Networking

JIAN LI

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

(Future Networked System)

Supervisor: Stefan Weber

August 2019

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Li Jian

JIAN LI

August 14, 2019

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Li Jian

JIAN LI

August 14, 2019

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. Stefan Weber, for his excellent guidance, innovative thinking, and endless patience. I would also like to extend my gratitude to my future girlfriend, who has not shown up till now, so that I have more time and energy to complete this study.

JIAN LI

University of Dublin, Trinity College
August 2019

An OpenFlow-based Software-Defined Networking Implementation over Named Data Networking

JIAN LI, Master of Science in Computer Science

University of Dublin, Trinity College, 2019

Supervisor: Stefan Weber

The Software-Defined Networking (SDN) architecture splits the control plane from the data plane, which uses a centralized controller for routing decisions and network management, while the switches are only responsible for data forwarding. SDN makes data forwarding more efficient and network management more convenient.The Named Data Networking(NDN) is “content-centric”, “destination-driven” and “connectionless-oriented” network architecture, which is designed to replace IP networks to offer fast, stable, secure, low latency, high throughput, mobile network services.Although both NDN and SDN are designed to improve the performance of the Internet, they are not competitive. It is possible to integrate these two architectures and combine their advantages.

This study analyzes the feasibility of integrating SDN and NDN architectures, as well as the characteristics of the existing integration solutions, and designs a new solution for migrating OpenFlow-based SDN to NDN environments. By deploying and evaluating the prototype of this model, it has been proved that this model can successfully integrate NDN and SDN, and it also has excellent scalability, compatibility, and security.

Summary

This study aimed to port OpenFlow-based Software-Defined Networking (SDN) architecture to Named Data Networking (NDN) and combined their merits. Researches on the current state of the art in the fields of SDN and NDN revealed the feasibility and necessity of integrating NDN and SDN. Moreover, there are many integration solutions for these two architectures that have been proposed and validated, but they either rely on traditional IP networks or have compatibility issues.

This research project referred to the traditional, IP-based OpenFlow protocol, and designed a new SDN protocol, which has similar functions to OpenFlow and can run directly on the NDN environment. Meanwhile, it also supports traditional IP routing management. Therefore, this model solves the dependency and compatibility issues of existing solutions. In addition, since the model runs over NDN, it benefits from the advantages of the NDN architecture, such as ”content-centric” security and efficient forwarding, it also provides services of network management and routing management to NDN.Apart from the comprehensive design of the model, including architecture design and protocol design, the project also developed a prototype and deployed it to the NDN simulation environment for evaluation. The results of the evaluation prove that it has the ability to perform devices management and route management, as well as excellent scalability, compatibility, and security.

Finally, the limitations of this model and the direction of its future work are also discussed at the end.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the recent decade, the increment of network traffic is enormous both in the type and the quantity. To a great extent, this has been fueled by massive mobile devices accessing the Internet. A large number of online services and cloud services have caused network data to become more diverse and complex, making traditional networks overwhelmed. Moreover, the demands for next-generation networks are critical, which must be faster, higher throughput, lower latency, and more robust than traditional IP Internet network.

Although it is challenging, the problems need to be solved eventually. There are two main directions for solving these problems, one is to adjust the existing network architecture, and the other is to design a new and more suitable network architecture.

As a result, Software-defined Networking(SDN)[1] and Network Functions Virtualization(NFV)[2] have been introduced as the former solutions. SDN is designed to improve forwarding efficiency and simplify network management, and NFV is used to improve network flexibility and reliability. These technologies can significantly improve the performance of current network constructions to satisfy the requirements of future next-generation networks.

On the other hand, Information-Centric Networking (ICN)[3] is regarded as one of the most promising future Internet architecture, which is designed to replace the current TCP/IP network architecture. This new network design has higher data forwarding efficiency, which can meet the demands of future network applications. It is also far superior to the current TCP/IP network in terms of mobility, security, and scalability.

Although both solutions have many advantages, they are still in the design and development phase. There are still many areas need to improve. This thesis is based on these new technologies and aims to improve its feasibility and increase its functionality.

This chapter outlines the background of SDN and NDN, and gives the project motivation. Then the aims and objectives are listed out followed by the study contribution. After that, the terminology section gives some necessary terms related to this study. The document structure is displayed at the end.

1.1 Background

This section explains the basic concepts of SDN and NDN. In order to reveal their advantages more clearly, the comparison with traditional network architecture will also be listed here.

1.1.1 SDN Background

In traditional network, routers and switches are the essential elements of data forwarding. They are called network nodes. Each node consists of two key components, the control-plane and the data-plane. The control-plane is responsible for routing decision, and the data-plane is in charge of data forwarding. Every node in a network calculates the routes to the data's destination and forwards data out of their interfaces independently. In other words, each node has to compute routes for data by themselves before forwarding. In addition, the transmission delay is also remarkably increased. Furthermore, the network scalability and manageability are very weak, since each node makes decision separately. If the scale of a network is large, each network policy change might mean a large number of network nodes need to be reconfigured. It becomes extremely difficult for the management of large scale networks.

Software-defined Networking (SDN) originated from the idea of solving these problems. As early as 2004, the Internet Engineering Task Force (IETF) tried to decouple the control and forwarding functions[4]. The OpenFlow protocol was proposed as a communication protocol in 2008, which can access the forwarding plane of a network switch or route over the network [5]. Until 2011, the Open Networking Foundation

(ONF) began to manage the standard of OpenFlow and released the first version 1.1[6]. Until now, the OpenFlow has been upgraded to v1.4[7].

SDN is designed to separate control-plane from data-plane of the network node[8]. This model abstracts the control-planes from all the nodes and put them to a centralized server called controller. OpenFlow is the protocol used by the nodes and the controller for communicating and exchanging information. So that all nodes in a network only have the data-planes and they are only responsible for forwarding data packets, no need to calculate routes for data since they do not have the control-planes. Instead, the controller takes the job of routing decision and instructs the nodes to forward data.

Some of the concepts here need to be precise. The data-plane is responsible for processing packets with local forwarding state. The forwarding decision in data-plane is determined by the forwarding state and packet header. The control-plane is used to set the forwarding state for data-plane. In the traditional network architecture, the control plane and the forwarding plane are integrated. SDN abstracts the control-plane and centralizes it to a control center to perform its functions. SDN is a set of abstractions for control-plane, not specific set mechanisms. OpenFlow is a standardized interface to switch, which is a vendor-independent protocol for communication between the SDN controller and switches.In addition to OpenFlow, there are other SDN protocols. The “SDN” concept mentioned in this paper refers to the general meaning in the usual sense. And we define “OpenFlow-based SDN” or “OF-SDN” specifically for SDN that uses the OpenFlow protocol.

In detail, for a node, it forwards data according to a table called FlowTable, which is maintained according to the controller’s instruction via OpenFlow protocol. FlowTable consists of a collection of FlowEntries, which indicates how to deal with a specific packet based on its flow information. If there is no FlowEntry for a packet, the node could send it to the controller directly or drop it. On the other hand, the central controller has two interfaces, the southbound and the northbound. Southbound is used to communicate with the nodes via OpenFlow protocol, and the northbound is responsible for connecting to upper applications. Upper applications do the actual network management and routing calculation. The controller collects network information from nodes for upper applications and translates the instructions from upper applications to the nodes. Figure 1.1 shows the comparsion of TCP/IP and SDN architectures.

To summarize, in OpenFlow-based SDN model, the network nodes only focus on

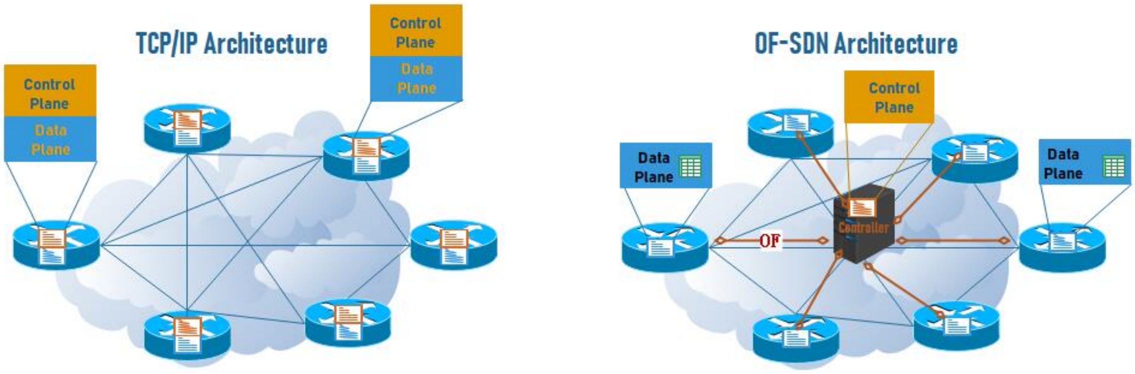


Figure 1.1: TCP/IP Architecture versus SDN Architecture

forwarding data, and the controller and upper applications are in charge of network management and routing calculation. Comparing with the traditional, distributed network architecture, OpenFlow-based SDN model is more efficient and has better maintainability and scalability. The detail information about SDN is included in Section 2.1.

1.1.2 NDN Background

The concept of Named Data Networking (NDN)[9] was derived from early Content-Centric Networking (CCN) project, which was first published by Van Jacobson in 2006 [10]. Since this data-oriented and information-centric network architecture proposed a new and fundamental solution to those unsolved problems in the current IP network, it has attracted the attention of industry and research institutions. In 2012, The Internet Research Task Force (IRTF) established a research group about information-centric networking (ICN)[11]. ICN was inspired by CCN mechanism and aimed to design an Internet infrastructure, which is data-centric, location independent, connectionless-oriented, in-network caching supported, secure, efficient, and robust. Till now, the ICN concept has become a broad research direction of content-centric or information-centric approach of network architecture, which is a general concept in this area; whereas CCN, NDN, the Network of Information(NetInf), and Publish/Subscribe Networking(PSIRP) are different proposed architectures of ICN[12].

NDN is an instance of ICN, and it is NSF-funded (National Science Foundation)[13].

In the beginning, the NDN project initially used the codebase of CCNx, but since 2013, it has forked a new version which is not related to CCNx any more[12]. Until now, there are a large NDN research group including sixteen NSF-funded principal investigators in twelve campuses and more than thirty institutions. Researchers are still working on the development of new features for NDN, including mobility support, real-time support, IPV6 support, and IoT support[14].

Comparing with traditional TCP/IP network architecture, NDN has many advantages. Firstly, it is DNS-exempted since data is forwarded according to its name, instead of any address. Secondly, Data transfer is connectionless, so it keeps away from many security issues, such as address spoofing, DoS, man-in-the-middle attacks. Thirdly, multicast support naturally makes data forwarding more efficient and saves a lot of bandwidth resources. Lastly, the in-network storage feature reduces latency and also saves bandwidth resources. Figure 1.2 shows the comparsion of TCP/IP and NDN architectures.

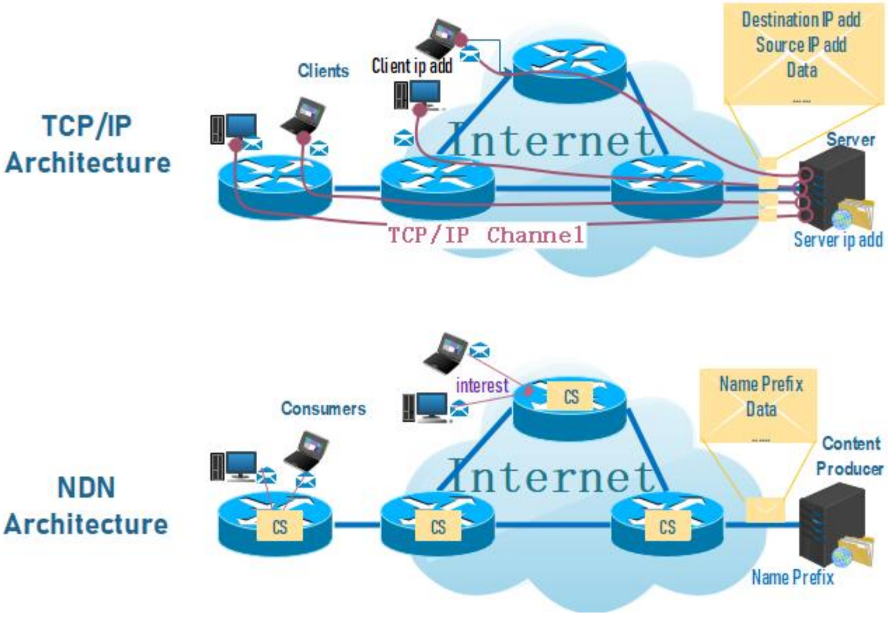


Figure 1.2: TCP/IP Architecture versus NDN Architecture

In conclusion, NDN is a promising technology that has many advantages over traditional TCP/IP networks, and it addressed many of the complex problems in TCP/IP networks. It is more suitable for today's big data network applications. The detail information about NDN is included in Chapter 2.2.

1.2 Research Motivation

We introduced the NDN and OpenFlow-based SDN concepts in the previous section and outlines their benefits. We regarded the OpenFlow-based SDN as a centralized and efficient network structure and the NDN as the final solution to the problems in the TCP/IP network. However, it seems that they are entirely two different technologies and no relationships between them. In fact, there is no conflict between the two protocols, and they can be integrated together and take both their advantages. In section 2.4, we do the research of feasibility of combining these to architectures in detail. In brief, OpenFlow-based SDN separates the control-plane from data-plane of the network node, but it just provide a tool to address network management problem[15] and only support TCP/IP network at this moment; while NDN is designed to replace the TCP/IP network, but it does not support centralized and programmable control. We believe that there should be an approach that runs an OpenFlow-based SDN liked model over NDN, and this model has the ability to manage NDN nodes and provide data forwarding information for them. This research aims to design such an architecture model.

1.3 Research Aims and Objectives

This research project aims to figure out an approach to integrate the OpenFlow-based SDN and NDN architectures, and combine their merits. By study their principles and existing integration proposals, we aim to design a SDN model which can run over NDN directly, and offer the functions of centralized management and programmable interface.

In particular, this research aims:

- To study the principles of OpenFlow-based SDN and NDN, and find out the feasibility of integrating these two paradigms.

- To design a prototype to achieve the essential functions of OpenFlow-based SDN over NDN.

- To test and evaluate the model's functions and performance.

1.4 Research Contribution

This thesis firstly analyzes the principles of OpenFlow-based SDN protocol and NDN architecture in detail and points out the feasibility of running OpenFlow-based SDN on NDN network. Based on this analysis, a new OpenFlow-based SDN model is designed and developed. Then it details how this model is deployed and implemented on NDN network, followed by the test of the functionality and evaluation of its performance. At the end of the thesis, it summarizes the advantages of this design and discusses the direction of future work. This research project demonstrates the feasibility of integrating OpenFlow-based SDN and NDN. It also presents the details of model design and development. This project can be a reference for migrating traditional and useful applications to the emerging NDN.

1.5 Document Structure

The structure of this thesis is as follows. Chapter 1 introduces the fundamental concepts of OpenFlow-based SDN and NDN, plus some background information about these emerging technologies, followed by the research motivation, objectives, and contribution. Chapter 2 analyzes the principles of OpenFlow-based SDN and NDN in detail, and researches the states of the art in these fields. Then the investigation of existing solutions is presented, followed by the feasibility analysis of integrating SDN and NDN. Chapter 3 states the problems and proposes solutions. The challenges are listed as well. The design of the model is presented in Chapter 4, including architecture design, functional design, message design, and approaches to address challenges. How to deploy this model is introduced in Chapter 5. The deployment environment and deployment method are the key points. A comprehensive evaluation is described in Chapter 6, which includes the evaluation contents, methods, process and results.

Chapter 7 summarizes this research project and outlines some limitations and future work.

Chapter 2

State of the Art

This chapter focuses on analyzing the principles and latest research results of OpenFlow-based SDN and NDN, and then investigates the existing fusion implementations of these two protocols and their contributions and limitations. Based on the analysis results, it also discusses the feasibility of integrating these two architectures.

In brief, this chapter includes sections as follows:

1. OpenFlow-based SDN architecture
2. NDN principle
3. Investigation of existing fusion models
4. Research of feasibility of combination
5. Summary

2.1 OpenFlow-based SDN

As introduced in chapter 1, Software Defined Networking (SDN) is an abstract model for data switching [16], which provides forwarding decision information and management function for transit devices in the network. In general SDN definition, it offers a programmable interface for upper applications via northbound and manages the switches via southbound. There are different protocols for the switch management. OpenFlow is one of them which has been widely supported. In this research project, we mainly investigate the OpenFlow protocol and use it as our model's reference. This section describes the OpenFlow-based SDN in detail.

2.1.1 OpenFlow-based SDN Components

This section investigates the OpenFlow-based SDN components and their functions. The main components are OpenFlow switch, OpenFlow controller and OpenFlow protocol.

- **Switch.** OpenFlow switch is the significant component in OpenFlow-based SDN architecture, which is responsible for data forwarding. Comparing with the traditional network node, such as router or switch, an OF-Switch only has data-plane but no control-plane, instead, there is a switch agent. Because of the lack of control-plane, the OF-Switch will not calculate routes for any data. As a result, it saves the computational resource and focus on data forwarding, the data forwarding efficiency in the whole network is much higher than traditional network.

  An OpenFlow switch consists of four elements: A FlowTable, a switch agent, the OpenFlow Protocol, and a Secure Channel. The FlowTable includes many FlowEntries, which indicate how to deal with a specific data flow. The switch agent communicates with one or more controllers by OpenFlow protocol and speaks to the data-plane to maintain the FlowTable. The OpenFlow protocol is the communication method between the switch agent and the controller. The secure channel connects the OF-Switch to a remote controller and transmits OpenFlow messages. Figure 2.1 displays the OpenFlow switch elements.

  Figure 2.2 shows the packet processing in OF-Switch. After the packet arrives, the in-port counter increases, and the key will be extracted. The key indicates the flow information, such as IP address, MAC address, TCP port, or VLAN ID. Then the switch searches its FlowTable against the key and finds the matched FlowEntries. There might be more than one matched FlowEntries. The first one which is fully matched will be selected. After that, the action associated with that entry will be applied on the packet. The action could be various, such as "forward out from an interface", "forward to the controller", "flood to all

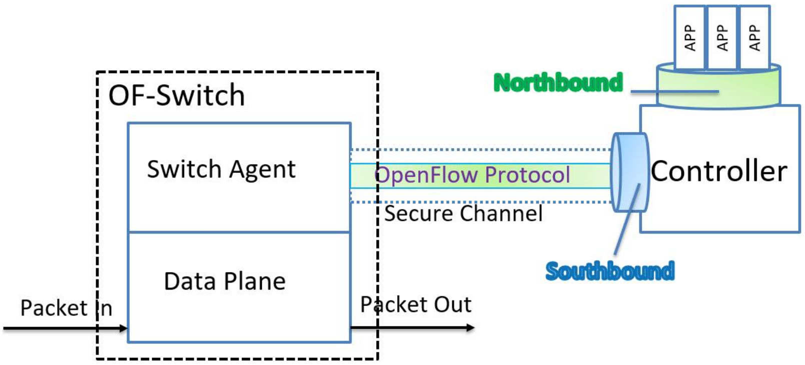


Figure 2.1: OpenFlow Component

interfaces except the incoming one", "drop", or "modify the packet head". If the packet head is changed, it needs to go back to search the FlowTable again and apply the corresponding action.

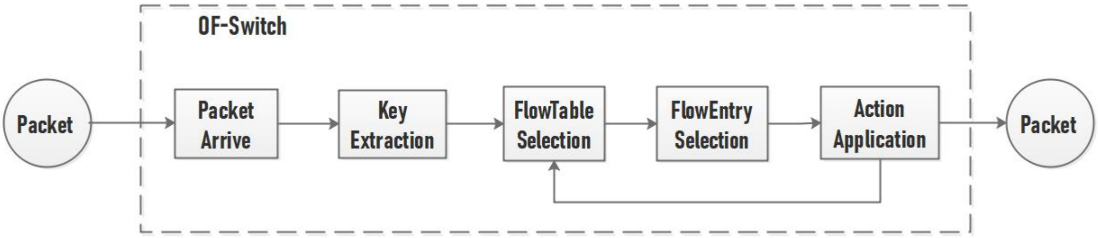


Figure 2.2: Packet processing in OF-Switch

- **Controller.** OpenFlow controller is another significant component in OpenFlow-based SDN architecture, which is responsible for communicating with OF-Switches and upper applications. An OpenFlow controller has two interfaces, the southbound and the northbound[17]. It interacts with OF-Switch agent by OpenFlow protocol from the southbound interface and communicates with upper applications from northbound. Figure 2.1 presents the controller components.

  The OF-Controller could collect the switches' information, including switch features, status, adjacency, and statistic information. The upper applications can use this information to monitor and manage network devices, calculate routes, and adjust policies. When the commands from upper applications come down to the controller, it translates them into OpenFlow message and sends them to switches. Because of its function, it can be regarded as an agent of upper management applications or a protocol translator.

It needs to be clear that the controller itself does not do any routing calculation or decision. It only collects information for the upper applications and conveys the decisions for the upper applications. In fact, the routing decisions and network management are done by the upper applications[18]. There are various upper applications and controller implementations, such as POX/NOX[19], OpenDaylight[20], ONOS[21] and Beacon[22].

- **OpenFlow Protocol.** OpenFlow is one kind of SDN protocols used for communication between controller and switches in SDN architecture. OpenFlow is maintained by Open Networking Forum (ONF)[23], and it has been upgraded to version 1.4 til now[7]. Figure 2.3 shows the normal implementation of Openflow protocol.

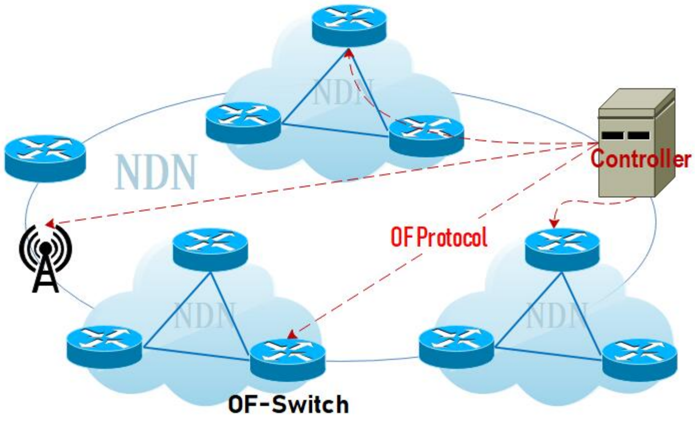


Figure 2.3: OpenFlow Implementation

In [24], OpenFlow is divided into four elements, the message layer, the state machine, the system interface, and the configuration. These mechanisms ensure that the switches and controller could understand each other's intention.

In particular, the OpenFlow protocol is only spoken by OF-Switch's agent and the controller via a secure channel. Since it is a vendor-independent protocol and compatible with different brands of devices, it is widely used in the industry.

2.1.2 OpenFlow-based SDN Implementation

Software-Defined Networking (SDN) is designed to improve data forwarding efficiency and simplify the network management. OpenFlow is one of SDN protocols, which is widely applied in industry and academia. This section outlines the OpenFlow-based SDN implementation.

OpenFlow protocol consists of a collection of messages[7], which is used for information exchanging between controller and switches. These messages can be grouped into three categories, as follows:

- **Controller-to-switch messages**, which are initiated by the controller and used to manage or inspect the switch directly, such as features message, configuration message, modify-state message, packet outs message, read-states message, barrier message, role-request message, and asynchronous-configuration message.
- **Asynchronous Messages**, which are initiated by the switch and used to notify the controller the status changes or network events, such as packet-in message, flow-removed message, port status message, and error message.
- **Symmetrc Messages**, which can be sent by either the controller or the switch and without solicitation, such as hello message, echo message and experimenter message.

The connection between the controller and the switch use TCP through a secure channel. Typically, the switch initiates the connection to the controller's IP address and TCP 6633 port. After the connection is built, both sides send hello messages and negotiate the version. Finally, the OpenFlow protocol is running on both sides, and the connection status is kept for information exchange [17].

The controller will send the feature request message to collect the switch's feature and status information, which could be used for management or route calculation by upper applications.

Each switch has at least one FlowTable used for data forwarding. The switch agent component maintains this table according to the OpenFlow message from the controller, and report the status changes to the controller. It consists of many FlowEntries, which indicates an action to deal with the matched data flow. The controller can update the FlowTable reactively or proactively[25]. Reactive Flow Entries method means that every new flow packet will be sent to the controller to inquire how to forward it. While the Proactive Flow Entries method means that the controller indicates the switch how to forward a specific data flow in advance before the Data packet comes into the switch. The former can save buffer resources, but increase the delay, while the latter can directly forward data, but need more cache resources.

FlowTable is located in data-plane, so the switch uses it to forward data directly. Every FlowEntry includes a group of flow information as “classifiers” for flow matching and actions for data packet processing. Actions include “drop”, “flood”, “forward”, or “modify”. One data packet could be processed when it gets in an interface and departs from an interface. Figure 2.4 gives the structure of FlowTable.

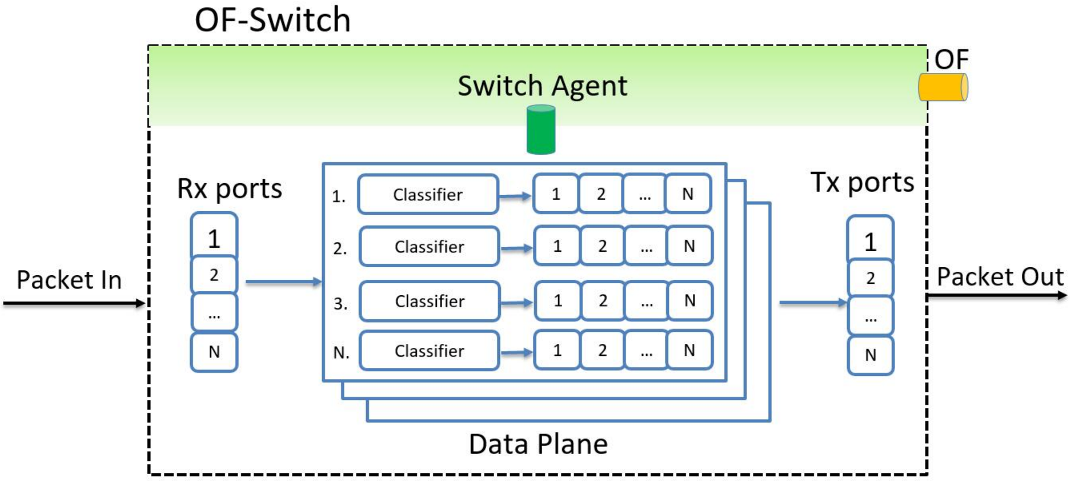


Figure 2.4: OpenFlow Switch FlowTable

2.2 Named Data Networking

Section 1.1.2 gives the background of Named Data Networking (NDN). This section describes more details about NDN, including the technology and the NDN developing project.

2.2.1 NDN Architecture

Since Jacobson introduced the CCN protocol stack in [10], all subsequent ICN forks have adopted this "thin waist" protocol stack, including CCN, NDN, Netinf, and PSIRP. It inherits the advantage of the IP network's "thin waist" model, which is that the simplicity of the network layer and less reliance on the unstable layer 2. Additionally, it makes fewer demands on under layers. The network layer in CCN, it is called content chunks layer here, can be placed over anything, even IP network. This "thin waist" architecture is also called hourglass architecture in NDN[26]. Even the NDN hourglass architecture derives from the IP network, but they are fundamentally different. In NDN, data delivery is consumer-driven, accompanying the built-in security and benefiting from its in-network storage. So the NDN architecture is more suitable for big data dissemination, network load balance, mobility, and low latency applications.

The NDN design team defined six principles to guide the design of NDN architecture [27]. So far, the basic structure has finished, and it is still in the process of improvement.

2.2.2 NDN Packets

NDN is a consumer-driven network architecture, which means that data users need to "express" their interests to some data first, and then their interests can be satisfied by the provider. So there are two types of packets, Interest and Data[10]. They contain the same name prefixes, and they are paired, which means the purpose of an Interest packet is getting a Data packet, and the Data packet can satisfy a corresponding Interest packet. To obtain information, a consumer needs to send out an Interest packet with the name of data it wants to retrieve. If the receiver of this Interest packet has the corresponding data, it responses this Interest packet with the Data packet.

The NDN Packet Specification[28] defines the packets' structures as showing in figure 2.5.

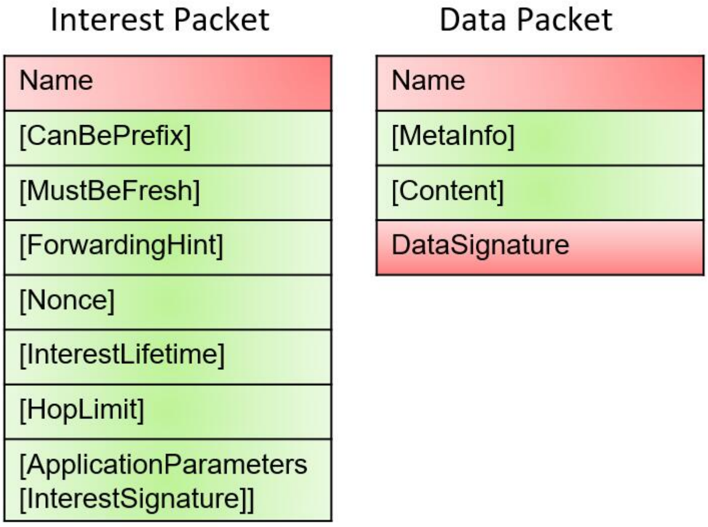


Figure 2.5: NDN Packets Structures

Some of these fields in both packets are required, and the others are optional. In this figure, the fields with the red background are required; the fields with the green background and enclosed name in square brackets are optional.

Each field is explained as below:

Interest Packet Fields

Field Name	**Type**	**Description**
Name	Required	The Interest packet's name, which is used to indicate the name of data the consumer wants, and the Interest packet is forwarded according to this name. Normally, the name format is hierarchical[10].

[CanBePrefix]	Optional	This field indicates whether the Interest packet should use the exact and full name of the interested data to fetch the Data packet. If it is not present, the Interest packet can use a prefix name, exact or full name to get the Data packet. Default Value = 0.
[MustBeFresh]	Optional	This field indicates if this Interest packet can be satisfied by Data which is stored in the Content Storage (CS) of a node and its FreshnessPeriod has not retired. Default Value = 0.
[ForwardingHint]	Optional	ForwardingHint field includes a list of delegations. These delegations represent data that can be retrieved by forwarding this Interest packet.
[CanBePrefix]	Optional	This field indicates if the name of the Interest packet is the prefix of a data's name or not. When it presents, and data with the name prefix the same as this Interest packet's name can satisfy this Interest packet. Default Value = 0.
[Nonce]	Optional	A randomly-generated 4-octet long-type string used for inspecting looping interests.
[InterestLifetime]	Optional	This field records the Interest packet's lifetime remains(approximately). The original default value is 4 seconds. The nodes it passed can modify this value and will drop it if its value is less than 0.

[HopLimit]	Optional	This field indicates the number of nodes the Interest packet can pass through. Its value range is 0-255. Every node will decrease its value by 1 if the value is great than 0, or drop it if its value less than 0. Note: This field is still under developing.
[ApplicationParameters [InterestSignature]]	Optional	This field is designed to carry any data as parameters for applications. If this field is filled, the digest component of the parameter needs to be added to the name prefix of this Interest packet. Note: This field is still under developing.

Data Packet Fields

Field Name	**Type**	**Description**
Name	Required	The Data packet's name. The Data packet is forwarded according to this name, and the packet will be forwarded to where the Interest packet with the same name (or prefix) comes. Normally, the name format is hierarchical.

[[MetaInfo]	Optional	This field consists of three optional elements: • ContentType, which includes the content type of BLOB, BINK, KEY, and NACK[29]. • FreshnessPeriod, which records how long a node should wait to mark the data as “non-fresh” after receiving this data. If the data is marked as “non-fresh”, it will not satisfy the Interest packet, which has set the fields of FreshnessPeriod and MustBeFresh. • FinalBlockId, which indicates the last block of a sequence of fragments. It is used in the case of data fragment.
[Content]	Optional	The real payload.
DataSignature	Required	The digital signature information of this data. It is required for security issues.

2.2.3 NDN Forwarding Mechanism

Jacobson first designed the link-state intra-domain routing mechanism in CCN[10], and Lixia Zhang extended it to NDN[9]. The details about the NDN forwarding mechanism are described in [30].

Before describing the NDN forwarding mechanism, an important concept, the “face” should be declared. The “face” in NDN has a similar mean of “interface” or “port” in the IP network. It represents the channel that can send or receive NDN packets. Since NDN can run over any network as depicted in 2.2.1, faces can be various types, such as an Ethernet port, a TCP connection, an IP next-hop, or an application. NDN’s “thin waist” architecture does not care much about the lower layer’s transmission methods, which is one of NDN’s benefits.

The Interest and Data packets are used to fetch and carry information. NDN forwarding mechanism is designed to forward these NDN packets. There are four essential components maintained by each NDN node for packet delivery: the Content Store(CS), the Pending Interest Table(PIT), the Forwarding Information Base(FIB), and the Forwarding Strategies.

- Content Store(CS). NDN nodes caches received data packets in CS for satisfying future Interest packets. Since NDN data is independent of the locations of the source and destination, the Interest packet can be satisfied by any nodes, so the closest one is the best. The NDN's in-network storage feature is based on this Content Store. A larger CS can cache more data, which means this node has the ability to satisfy more Interest packets by itself.

- Pending Interest Table(PIT). This table records all the Interest which has been forwarded out to fetch Data but not satisfied yet, and its incoming and outgoing faces. This information can prevent sending the same Interest packet more than once. In addition, it is also used to forward the Data packet to the correct downstream.

- Forwarding Information Base(FIB). A routing table indicates the next-hops for forwarding a specific name prefix of an Interest packet. To update this FIB may need information from other protocols or mechanisms, e.g., IP routing protocols or NLSR described in 2.2.4.

- Forwarding Strategies. A series of rules indicate how to forward the Interest packet. In another word, it decides which and how many next-hops will be chosen. E.g., the “Best Route Forwarding strategy” will only choose the lowest cost or delay next-hop, while the “Multicast forwarding strategy” will choose multiple next-hops.

The processes of Interest and Data packets in an NDN node are different. Figure 2.6[30] shows the differences.

- Interest packet process.
  When an NDN node receives an Interest packet, the CS will be checked first. If the matched data to this Interest(the name is the same) exists, the node will

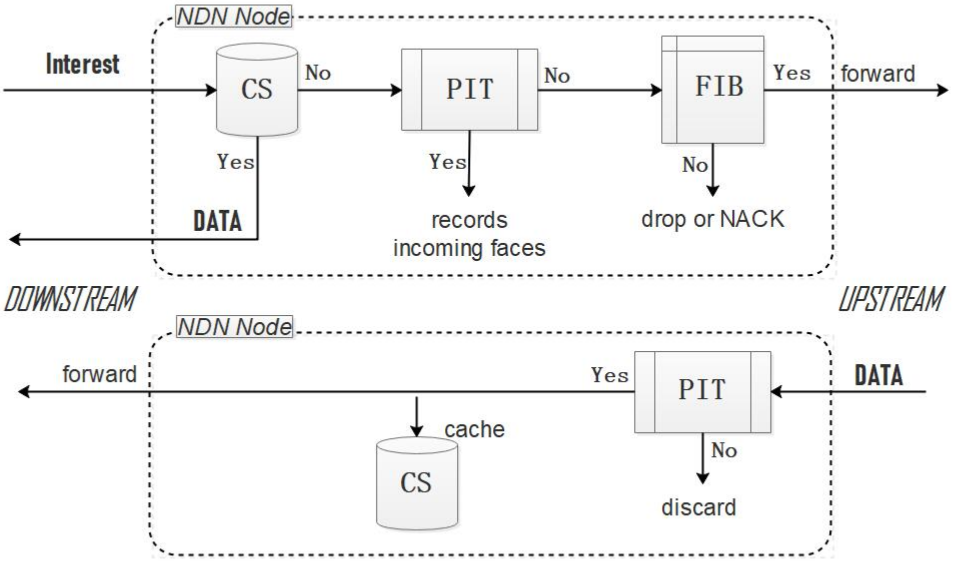


Figure 2.6: Interest and Data processing in NDN

send the Data packet directly to the face which the Interest comes. If there is no matched data, the PIT will be checked next. If a matching entry exists, which means the same Interest has been sent out and is still waiting for the Data packet, so the node will not forward this Interest packet again, and just records the incoming faces to PIT for satisfying it when data comes in the future. This feature is another NDN's benefit, naturally multicast support. If no existing record of this Interest, which means this is the first Interest packet for the name prefix. The node adds an entry to PIT to record the name prefix and the incoming face. After that, the FIB will be searched to forward this Interest. According to the FIB, the node finds out the corresponding next-hops for the prefix name, and sends the Interest to one or some next-hops. The number of next-hops is based on the Forwarding Strategies configured on this node.

At last, the Interest packet will be sent upstream. The other nodes in NDN do the same thing until the Interest is satisfied.

- Data packet process.
  When a Data packet is received, the node retrieves its PIT to find out the records about the related Interests, then forwards data to the faces according to the PIT

records and then delete this records from PIT. Meanwhile, the data will be stored in its CS for future usage. If there is no entry found in PIT, which means all of the Interest packets are timeout, the node will simply store data in its CS.

In this way, all the Interests with the same name who arrived this node will be satisfied, including the future coming Interest. Moreover, the Data packets always take the reverse path of Interests. It is the third benefit of NDN.

In general, the packet forwarding mechanism in NDN does not relate to any kinds of addresses. The Interest packet is forwarded according to its name, and the Data packet is delivered based on the PIT state information set up by the Interests on each hop. This design can avoid many kinds of problems existing connection-oriented networks, such as TCP/IP network. Furthermore, its high forwarding efficiency stems from this mechanism design, such as the naturally multicast support, the in-network store, and the Interest aggregation. It is the main advantages of NDN over current IP network.

2.2.4 NDN Implementation

NDN project has been started since 2010[31] and aims to develop a new data-centralized Internet architecture based on NDN architecture. It uses a testbed to deploy, simulate, and analyze the proposed architecture. Until now, the project has developed specifications and prototype implementations of NDN protocols and applications. NDN Platform 0.6.5, the "bleeding edge" version, is the latest released version[32], which gathers many new components, including the most significant ones, the NDN Forwarding Daemon (NFD), and the Named Data Link State Routing Protocol(NLSR). NFD and NSLR are the main NDN implementation parts.

NDN Forwarding Daemon (NFD)

NDN Forwarding Daemon(NFD) is an open-source NDN network forwarder developed by the community of NDN project[33], and it has become the core component of the NDN Platform. The early version of NFD referred to the CCNx software package, but now it is not in any part derived from CCNx codebase and may not be compatible with CCNx any more. It evolves with NDN protocols and additional features. Its modular design has excellent extensibility so that it supports diverse experimentations of NDN technology. The NFD team has also maintained a developer's guide, which can give tips for external contributors[34].

NFD is developed via C++ and can be run on Linux system. It maintains six major modules to implement the NDN node's functions[33]. It can be deployed independently or use NDN simulator, which has integrated the latest NFD version.

Named Data Link State Routing Protocol(NLSR)

Named Data Link State Routing Protocol(NLSR) is a routing protocol in NDN which is developed by NSF-sponsored NDN project team. It is an open and free software packet, and the first version was developed in 2013[35]. Since 2014, it has been deployed on NDN testbed. The latest version is upgraded to 0.5.13[36], and many features are introduced.

As discussed in Section 2.2.1, the NDN architecture can run over other protocols, and it forwards NDN packets according to the name prefix and PIT status. However, the other information is required to update the FIB, which could come from any protocols or mechanisms, such as IP routing protocol. The NLSR is one of those protocols that can populate NDN's Routing Information Base(RIB).

NLSR is designed specifically for NDN by the NDN project team, which uses NDN's Interest/Data packets to disseminate routing information. This intra-domain link-state routing protocol provides many features that are different from IP-based link-state routing protocol, such as, hierarchical naming scheme, hierarchical trust model for security, using ChronoSync to update routing information, and a simple way to calculate multiple routes. The details of NLSR design is described in [37]. Its source code can be download from the official GitHub repository[38] and deployed on any NDN environment. The NDN simulator, Minindn, has integrated it.

2.3 Investigation of existing Fusion Models

SDN and NDN are emerging solutions to deal with the problems in the traditional IP network, and both of them have their own merits and limitations. So, Integrating SDN and NDN, and combining their advantages have become an exciting topic. Many organizations and researchers proposed solutions to fuse them together. This section depicts some of them.

The existing integration solutions can be grouped into two categories[39]: the short-term approach and the long-term approach.

For the short-term approach, the ICN/NDN is running over the traditional IP network, and the ICN/NDN packets are encapsulated by IP packet header. The SDN management protocols still use IP protocol and channel to manage the ICN/NDN network. This approach is considered as a transitional solution, which does not need to replace all of the devices in the current IP network, but it still relies on the IP network architecture, so the improvement of forwarding efficiency is limited.

For the long-term approach, the IP architecture will be completely discarded. SDN runs over ICN/NDN directly and uses the Interest and Data packets to achieve network management. Since the ICN/NDN is a clean-slate architecture, the compatibility issues could be the biggest challenge. So the whole system of the network needs to be redesigned, including the hardware of nodes, the ICN/NDN applications, and the SDN architecture and protocols. This approach can fully exploit the high forwarding efficiency of ICN/NDN and the management advantage of SDN. However, it is much costly and complicated.

Salsano's solution is an example of the short-term approach. Salsano *et al.*[40] proposed a method to integrate ICN with SDN in 2013 and updated the model in 2018[41]. They noticed that the shift from the traditional IP network to ICN could be costly since a large number of existing networking devices need to be replaced. Their solution is based on CONET(COntent NETwork)[42] and uses the NRS(Name Routing System) [43] to overcome the challenges of high delay and low throughput, which are caused by the processing of name-based routing calculation in SDN architecture. This approach exploits SDN architecture to deploy ICN over IP network. The NRS hosted on network device just like the switch agent of OpenFlow-based SDN, and its function is similar to the Domain Name System(DNS) in the IP network, which resolves the content names into next-hops. This model provides an available solution for converting legacy networks to ICN.

Signorello presented a representative example of the long-term approach. Signorello *et al.*[44] developed a preliminary open-source implementation of NDN with P4 (Programming protocol-independent packet processors)[45]. They programmed a software switch to match the NDN packet process. As the TLV (Type-Length-Value) packet of NDN is variable, they use a parser to split the TLV packet into several parts and resolve them. There is a count table designed to count the name elements which are used for matching FIB. The hashname table calculates the hash of full name and name prefix. They used a register to maintain the state of pending Interest packet, and PIT reads this register by name hash to determine how to match FIB. This model is much complicated, and all components are newly designed. At present, this kind of models have not a unified standard.

Table 2.3 compares the characteristics of these two categories of solutions.

Table 2.3: The short-term solution versus The long-term solution

Short-term Solution:	**Long-term Solution:**
Run over IP network	*Run over ICN/NDN*
Use traditional SDN protocol	*Design new SDN protocol*
Compatible with traditional devices and APPs	*Design new devices and APPs*
Clear architecture	*Complex architecture*
Mature standards	*No uniform standard*
Limited efficiency	*High efficiency*
Cheaper and easy solution	*Expensive and difficult solution*
Transitional solution	*Final solution*

2.4 Feasibility of Integrating SDN and NDN

As discussed in Section1.1, both SDN and NDN are designed to address problems in the current flat, distributed and host-centric network, but they use different ways and solve different problems. For SDN, it aims to offer a powerful centralized management system to improve the forwarding efficiency and simplify the network management with its global view of the network. NDN treats content as the network primitive and decouples it from the location. These two promising networking paradigms are not competing. In fact, they can absorb each other's advantages and improve themselves. For instance, SDN can offer NDN centralized network management, node deployment, strategy distribution, traffic monitoring, and programmable API. In return, SDN can benefit from NDN's in-network caching feature and data-centric security[9]. Therefore, in terms of functionality, integrating SDN and NDN is feasible.

NDN is a new network architecture, on which data transmission is no longer required to establish an IP channel in advance. In addition, its customer-driven delivery method is the biggest challenge to current SDN architecture, which needs real-time data interaction between SDN switches and controllers. As Section 2.3 describes, there are two kinds of solutions. The short-term solution is to run SDN on the traditional IP network to manage NDN nodes, so that the off-the-shelf SDN protocol and architecture can be deployed directly. The long-term solution is to design new SDN architectures and protocols which are able to run on the NDN directly. [39] investigates many representative solutions in both categories and reveals that they are feasible. So, from a technical perspective, integrating SDN and NDN is feasible.

In reality, integrating SDN and NDN is also feasible and necessary. Various solutions for migrating IP network to NDN have been widely proposed and validated[46][47][48], it may become a good candidate for future network architectures. If the NDN can benefit from the merits of SDN, such as centralized device management and routing management, then its availability, scalability, etc. can be improved. Both long-term and short-term integration solutions of SDN could provide a powerful, vertical, layered, centralized, and programmable management model for NDN.

In short, NDN and SDN integration is feasible in terms of functionality, technology, and practicality.

2.5 Summary

This chapter firstly details the principle of OpenFlow-based SDN, including the main components and features, as well as the implementation of OpenFlow-based SDN. And then it describes the NDN architecture, the packets types and fields, and NDN forwarding mechanism, along with its implementation. After that, the investigation of existing integration solutions is presented, followed by the final feasibility study, which reveals that the fusion SDN and NDN are technically, functionally, and practically feasible.

Chapter 3

Problem Statement

After analyzing the technical principles of NDN and OpenFlow-based SDN, investigating the others' solutions of integration, and researching the feasibility of fusion, it has become evident that the combination of SDN and NDN is necessary, and its benefits are discussed in the previous chapter. This chapter outlines the problems of the integration and emphasizes our design's innovation. At last, the challenges of our solution encountered are also presented, followed by a concise summary.

3.1 Abstraction of the Problems

As stated in the previous chapter, many solutions for the fusion of SDN and NDN have been brought out, and they can be grouped into short-term solutions and long-term solutions. Many of them have been evaluated to prove their feasibility. However, they all have their own limitations.

For short-term solutions, they use the traditional IP network to run SDN to manage NDN nodes, e.g., [49]. This solution can directly use the traditional SDN architecture for NDN network management, and it can also be compatible with traditional IP network management. But it still relies on the IP connection, can not benefit from the features of the NDN network, and it makes the original IP network more complicated. This solution can only be a temporary transition plan.

For the long-term solutions, they are completely redesigned management tools which can run over NDN. But they are not directly related to the current SDN, nor can they be compatible with the current SDN architecture. Too complicated is also the shortcomings of this kind of solutions, e.g.,[40].

Our goal is to find a compromise that can be run directly on the NDN for centralized network management, and it is also compatible with current SDN architectures and protocols, such as the OpenFlow protocol. Since OpenFlow is a wildly supported SDN protocol nowadays, we design that our model can run over the NDN and adopt the OpenFlow architecture, which is compatible with traditional OpenFlow SDN.

3.2 Improvement and Innovation

Our design model is entirely independent of the IP network architecture so that it can be seen as a long-term solution. Moreover,

we refer to the traditional OpenFlow protocol and try to inherit more original structure and packet type. In FlowTable, the field supporting IP packets are reserved. So, our model structure is clearer and more compatible.

Compared to other short-term solutions, our design does not rely on IP networks, so it can fully benefit from the advantages of the NDN network without the bottleneck of the IP network. Also, compared to other long-term solutions, although the OpenFlow architecture is redesigned, we referred to the traditional OpenFlow structure, the message types and formats. We also reserved some fields in data packets for supporting the tradition OpenFlow protocol. As a result, our model has better compatibility.

In short, compared to other solutions, our design not only benefits the characteristics of the NDN network, but also the architecture is more explicit and has better compatibility.

3.3 Challenges

Traditional OpenFlow SDN is based on IP networks, which uses IP connection to exchange information. The controller and switches talk with each other directly so that the controller can query the status of the switch in real-time and send commands at any time. However, the NDN is totally different. Data delivery is customer-driven, like publish/subscribe paradigm[50]. In addition, because of the in-network cache mechanism, the customer and producer cannot “call” each other directly. All in all, porting OpenFlow-based SDN to NDN is not easy. This section summarizes the main challenges here.

1. NDN Interest packet cannot carry payload.
   Since NDN data delivery is customer-driven, the data producer cannot send data to the consumer directly. Instead, the consumer has to send an Interest packet first to fetch data it wants to retrieve, then the producer reply this Interest with data. Unfortunately, the Interest packet is designed only to fetch Data packets, which correspond to the name listed in the Interest packet. It usually cannot carry any payload. However, in our model, if the communication between controller and switches always uses this publish/subscribe paradigm, the efficiency could be very low. We need to find a way to address this problem.

2. How could the nodes get up-to-date information from their controller?
   One of the most significant NDN's advantages is ”in-network storage”, that is, the Interest packet might be satisfied by any node if data in its ContentStore. So NDN naturally supports multicast. This feature is excellent in forwarding large data, but it causes a problem in our model. The consumer, it could be controller or switch, sends an Interest packet with the same name to query the newer version of the same data in the producer, but this Interest is always responded by the nodes which have the same name data but the old version in its CS. How can we send the request to the target and get the latest version data?

3. How do the switches or controller distinguish different types of messages if they have the same name prefix?
   For SDN protocol, different types of messages have specific functions. But in NDN, there are only two types of packets: the Interest packet and the Data packet. How can we use these two types of packets to send different types of SDN messages, and how does the receiver distinguish them? Especially when a node receives different types of messages with the same name prefix, how could the node recognize them?

4. How does the controller proactively send messages to a specific switch?
   Since the NDN's publish/subscribe paradigm, the “conversation” between two

nodes should start from one node by sending an Interest packet to the other's advertised name prefix. In our NOF-SDN model(the "NOF-SDN" concept is defined in Chapter 4), the "conversation" is always initiated by switches, because the controller has a fixed and advertised name prefix for listening to incoming Interest packets. However, the switches do not have this fixed name prefix. So the problem is that if some issues happen and the controller needs to inform some switches immediately, e.g., the topology changes, how does the controller start the "conversation" to tell specific switches?

5. How does the NFD trigger the NOF switch agent's function?
   Our NOF-SDN model is based on NDN but an independent application. We do not intend to change anything of NDN and its underlying protocols, such as NFD and NLSR. So the problem is how to combine our model to the NDN. In other words, when an incident happens, how does the NDN node trigger our NOF-SDN switch agent's functions?

3.4 Summary

This chapter first outlines the problems of other solutions, that is, the short-term solutions rely on IP networks and the long-term solutions are not compatible with traditional SDN architecture. Then displays the innovation of our model, which can run on NDN directly and support the traditional SDN structure. Finally, the challenges in our design are also pointed out. The next chapter will describe how our design addresses these issues.

Chapter 4

Design

In the previous chapters, by studying the background information and status of the art about NDN and OpenFlow-based SDN, we investigated the existing solutions of integrating these architectures, and analyzed their advantages and disadvantages. We also figured out the existing problems. At last, we put forward our objectives and challenges.

Our purpose is to design a new OpenFlow-based SDN model, which can run over NDN directly and has the ability to manage the NDN nodes, makes the routing decision for the nodes. For the convenience of description, we define it as the NDN-based OpenFlow SDN(NOF-SDN) architecture, and define the protocol as the NDN-based OpenFlow(NOF) protocol. This mode is different from the original OpenFlow protocol, but it is referred to the traditional OpenFlow protocol architecture. The design process includes three aspects: the components architecture, the components function, and the NOF-SDN packets.

This chapter describes the design of the integration of NDN and OpenFlow-based SDN based on this dissertation's objectives. The first section is about the component architecture design, which includes the switch and controller architectures. The second section depicts the components function design, which presents the functions the switches and controller have and how they work. According to the structure and function, the NOF-SDN message design is introduced in section 3. The challenges raised in Section 3.3 are resolved in section 4. Finally, the last section gives a brief conclusion.

4.1 Component Architecture Design

Component Architecture design consists of switch architecture, controller architecture. The design details are included in this section. The “switch” and “controller” are the concepts in OpenFlow-based SDN model, while in NDN, the network forwarder is called “node”. Since our project is to port OpenFlow-based SDN to NDN, so we still use “switch” and “controller” to represent the NDN forwarder and controller.

4.1.1 Switch Architecture

As introduced in section 2.2.3, the Interest packet and Data packet in NDN are processed in different ways. For routing decision, we care more about the process of Interest packet since the Data packet is always delivered as the reverse path of Interest packet. Figure [111] shows the process of Interest packet in NDN nodes.

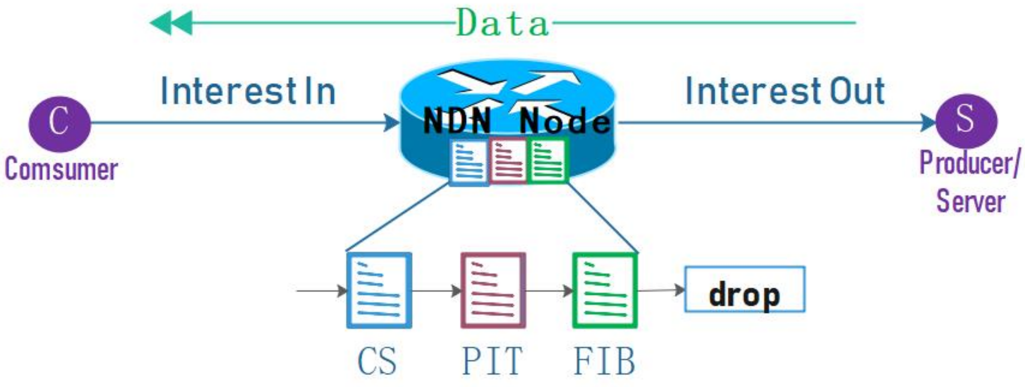


Figure 4.1: NDN Packets Process

After receiving an Interest packet, the NDN node searches its ContentStore first. If there is no matched data, then the PIT is checked. The presence of matched PIT entry means the same Interest has been forwarded; while the absence of matched PIT record means it needs to be delivered. Then the node will search its FIB to find the next-hops to send this Interest packet. If there is no corresponding route in FIB, the Interest packet will be discarded. The NDN node thinks itself has no knowledge or ability to forward this Interest packet.

On the other hand, the packet process in traditional OpenFlow SDN architecture is different. See Figure 4.2. Since the switch does not have control-plane, it will not calculate routes for any packet. It can only forward packets according to its FlowTable. So when a packet is received, the switch searches the FlowTable to find a matched FlowEntry, then forwards the packet as the FlowEntry indicated. If there is no matched FlowEntry, the switch will send an OpenFlow message to ask the controller how to deal with it. If no instruction receives, the node will simply drop the packet.

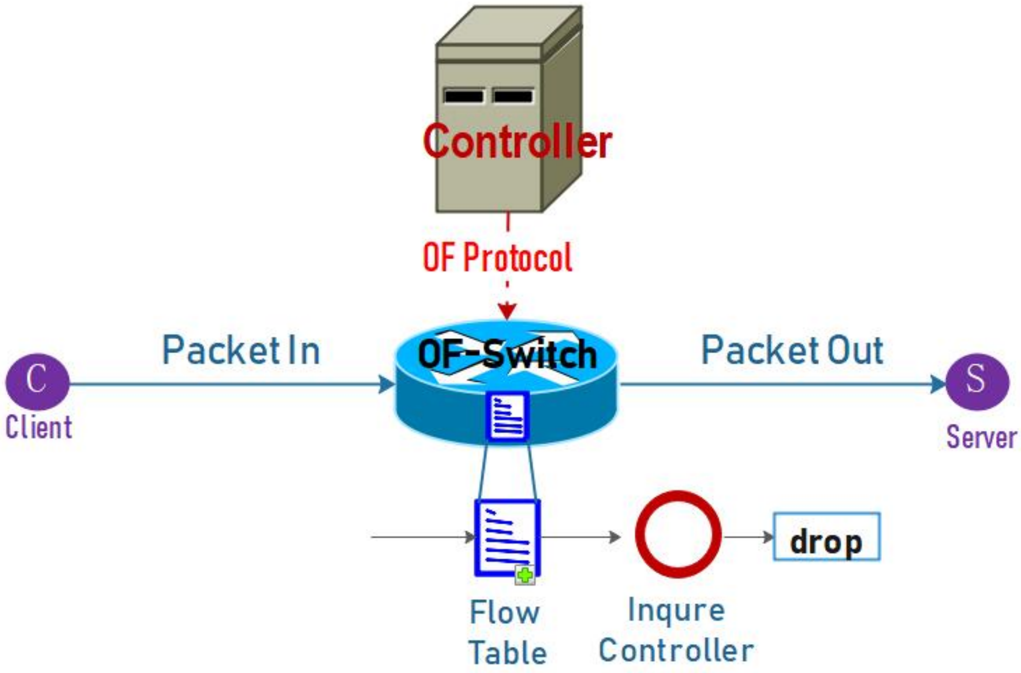


Figure 4.2: SDN Packets Process

The purpose of this project is utilizing the OpenFlow-based SDN structure to manage NDN nodes and to make routing decision for the nodes. By analyzing these two packet processing mechanisms, it is noticed that they can be combined. Figure 4.3 presents our design of the combination.

In this model, the packet processing method in OF-Switch is ported to the NDN node. As a result, in addition to the NDN packet forwarding capacity that the NDN node already has, it also can process the packet as an OF-Switch does.

To be precise, the OF-NDN switch in our model processes the packet as the same

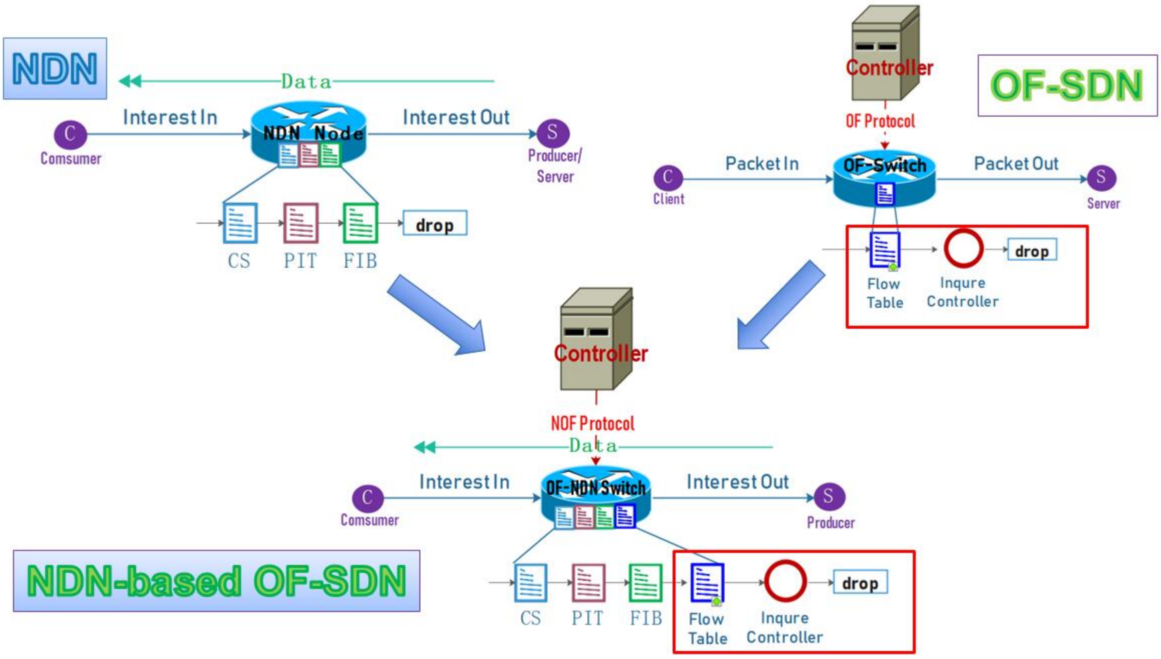


Figure 4.3: Combination NDN and OpenFlow-based SDN

as the normal NDN node does first. But if the packet cannot be forwarded since lack of matched FIB item, it will not discard it directly like a normal NDN node. Instead, it will continue to process it as an OpenFlow-based SDN switch does.

Specifically, we treat the traditional NDN node as a forwarding switch on the data-plane. On top of that, we added a Switch Agent that functions like the Switch Agent on a traditional OpenFlow-based SDN switch. The Switch Agent uses the NDN-based OpenFlow (NOF) protocol we designed to communicate with the remote controller, and the management channel to monitor and adjust the status of NDN node. In addition, it creates and maintains a FlowTable, as well as deals with the query requests. When the NDN node finishes processing the Interest packet as the traditional NDN node does but cannot find a matched entry in its FIB, it will continue to look up the FlowTable maintained by the switch agent and process the packet according to the actions included in the matched FlowEntry. Even though there is no matched FlowEntry in the FlowTable, the NDN node can also inquire the remote controller through the switch agent how to handle the packet. Moreover, the switch agent also maintains a database called Status Database(SDB), which is responsible for monitoring the NDN node's status and report to remote controller. Figure 2.3 shows the design architecture of our NDN Node. We define this NDN node as NOF-Switch.

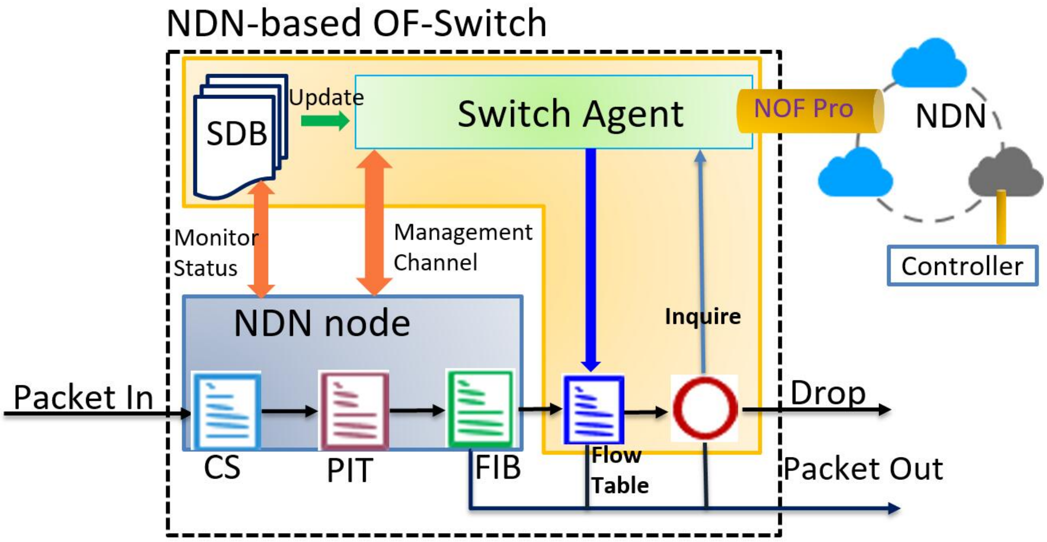


Figure 4.4: NDN-based OpenFlow Switch Structure

In our model, the FlowTable is used to forward the NDN Interest packets, not IP packets, so we redesigned the FlowTable. Considering compatibility, we reserved a field in FlowEntry for saving traditional IP flow information, so that this model has the ability to support the traditional OpenFlow protocol. If necessary, we only need to put the traditional IP flow information into the FlowEntry Ethernet Prefix field. Figure 4.5 shows the structure the FlowTable.

The FlowTable include s a set of FlowEntries, and each FlowEntry consists of Three areas: the Match Rules, the Counters, and the Actions. Each area has several fields, which are displayed in Table 4.1.

Table 4.1: FlowEntry Fields

Field Name	Description
Match Rules Area:	Used for matching items.

Ethernet Prefix	Reserved for traditional IP flow information.
Face	Incoming face.
Prefix	Interest name prefix.
Priority	Used when multiple items matched.
Counters Area:	Used for counting matched times,and lifetime.
Entry Counters	Record the matched times.
lifetime	Indicate the lifetime of this entry.
Actions Area:	Used for specifying the actions.
Action	The actions should be done if match this item.
Out Faces	The faces used for forwarding the packets.
Flags	Indicate if it needs to report to the controller when this item is timeout.

In terms of the Status Database(SDB) component, it is designed to check the current status changes of the NDN node periodically, including channels, faces, fib, rib. If there is an update, the switch agent will send a new version Hello packet to the remote controller and wait for the controller to fetch the new status information.

4.1.2 Controller Architecture

As standard, our controller has Southbound and Northbound, which are used for communication with switches and upper applications. The controller includes two main components: the Function component and the Databases. Figure 4.6 shows the controller structure.

As for Function component, it includes Listener, Checker, and Distributor. The Listener is in charge of listening to incoming packets and assigning them to different processes. The Checker is responsible for checking the status changes. And the Distributor is used for sending packets to different switches.

In terms of Databases, there are two databases: NPT and FDB.

The NPT represents Node Prefix Table, which records all the switches managed by the controller and their advertised prefixes. The content of each record is displayed in

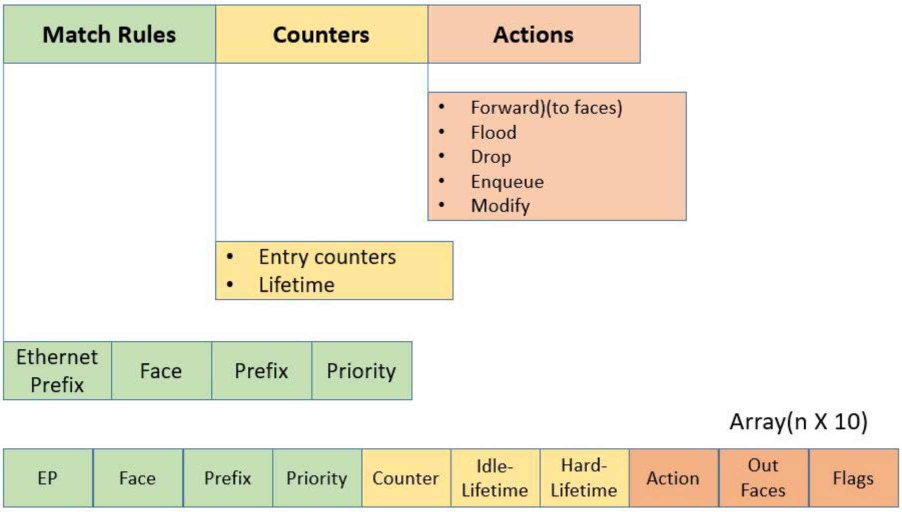


Figure 4.5: NDN-based FlowTable Structure

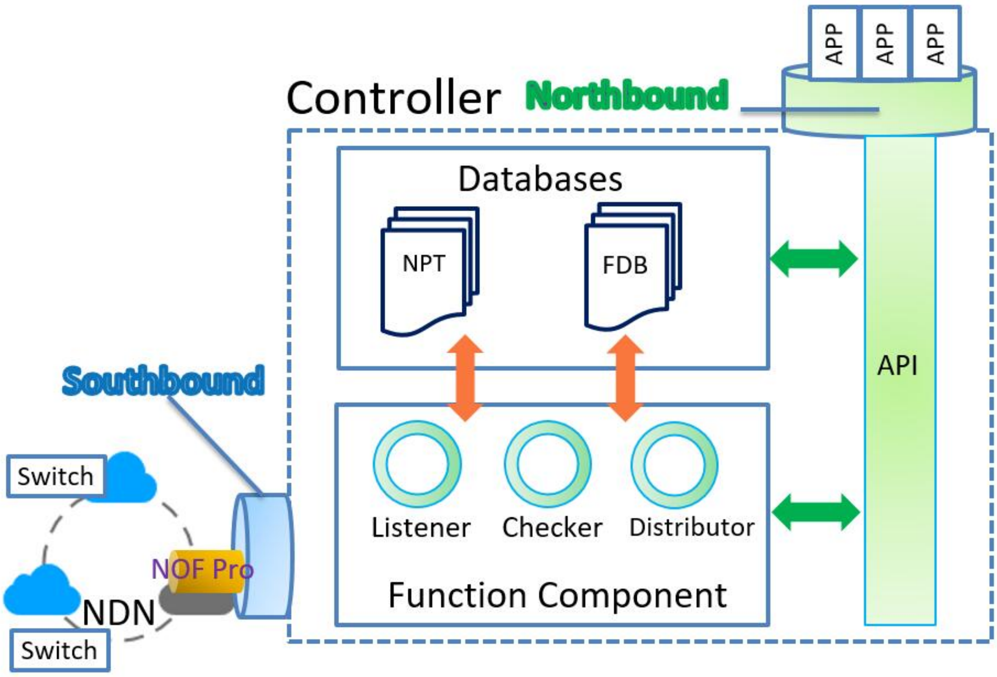


Figure 4.6: NDN-based OpenFlow Controller Structure

Table 4.2

Table 4.2: NPT Content

node_id	*version_number*	*type_number*	*node_prefix*

The *node-id* indicates the node; the *version_number* represents the newest version; the *type_number* is the NOF protocol version and message type; the *node_prefix* is the advertised prefix by the node, and controller can used this prefix to fetch the node's feature.

The FDB stands for Feature DataBase, which includes FIB database and Face database. Both of them store the node's features. The content of FIB database is displayed in Table 4.3

Table 4.3: FIB Database Content

node_id	*prefix*	*next_hop*	*cost*	*next_hop*
cost	*next_hop*	*cost*	*reserved*	*reserved*

The content of Face database is showed in Table 4.4.

Table 4.4: Face Database Content

node_id	*face_id*	*remote*	*local*	*congestion*
MTU	*counters*	*flags*	*reserved*	*reserved*

This information is maintained by NFD on each node. The controller fetches it to update the databases when new version hello packet is received. The process of obtaining this information is described in section 4.2.

4.2 Component Function Design

This section is divided into two parts, which portray the switch functions and the controller functions, respectively. In each piece, we highlight the processes for all use cases to clarify the operating approaches. Besides, some of the concepts in this section may not have been introduced in previous chapters, but they will be covered in later sections.

4.2.1 Switch Functions

In this model, four main functions are defined for the NOF-Switch: the Status Monitoring, the Configuration Fetching, the Topology Change Reporting, and the Route Inquiring.

The switch agent uses Status Monitoring function to detect changes of the NDN node's status, including the FIB, Faces, Channel, and RIB status. If any changes happen, the agent reports them to the controller, and then the controller fetches the new feature information to update its Feature Databases. The Configuration Fetching function module is designed for switch agent to get configuration from the controller. The Topology Change Reporting function module is used for reporting topology changes or errors to the controller. Also, the Route Inquiring function is responsible for asking the controller for route information when the NDN node cannot forward the Interest packet since the lack of FIB next-hop. Figure 4.7 shows the processing flow for all functions.

The specific processing flow of each function is as follows:

- Status Monitoring function
  This function module checks the status update of the switch periodically. When a status update is detected, it increments the version number of the HelloReq message and sends it to the controller, then waits for the response. If the HelloRes message is received, it will wait for the FeatureReq message sent by the controller and respond with its feature data. After that, the function continues to monitor the status.

- Configuration Fetching function
  From the switch starts, this function module sends a CtrlInfo Request message with a long lifetime to the controller and then keeps waiting for the CtrlInfo Response message, which contains the newest configuration information.After receiving the CtrlInfo Response message, it will send a new CtrlInfo request with a higher version number, and then continue to wait for the new response of the CtrlInfo request packet.

- Topology Error Reporting function

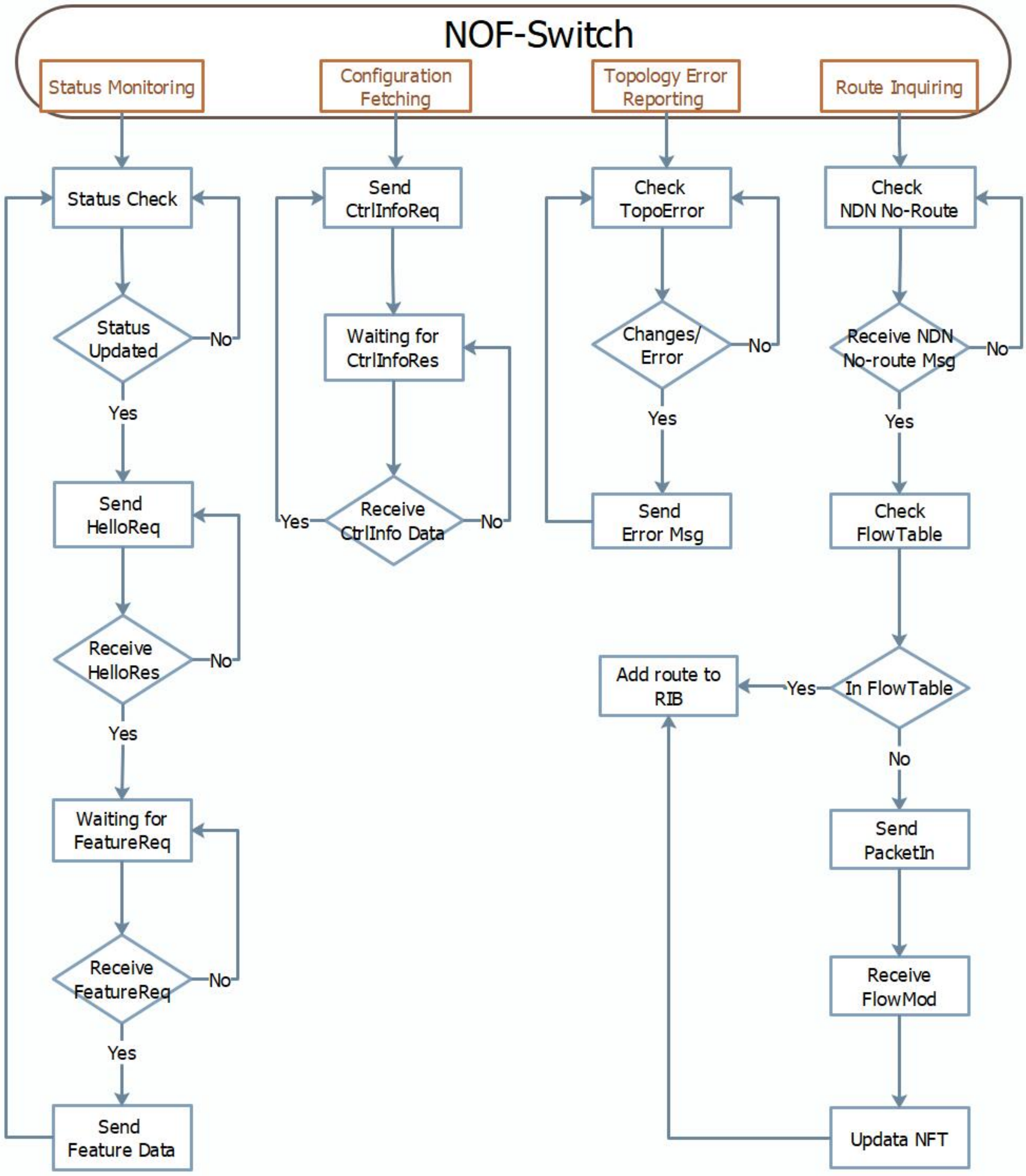


Figure 4.7: NOF-Switch Functions Flowchart

This function module is used to check topology errors periodically. The Error message will be sent to the controller if an error happens.

- Route Inquiring function

  The Route Inquiring function module monitors the NFD's "no-route" notification. If a "no-route" notification is received, it searches the corresponding record of the prefix in the FlowTable. If there is a matched record in the FlowTable, the next-hop will be added to the RIB directly; if no matched record is found, the PacketIn message will be sent to the controller for inquiring route information. And then the FlowTable will be updated according to the received FlowMod message from the controller; meanwhile, the next-hop will be added to the RIB.

4.2.2 Controller Functions

Corresponding to the functions of the switch, the controller also has four function modules: the Feature Updating, the Configuration Distributing, the Topology Error Operating, and the Route Responding.

The Feature Updating function is designed to fetch new features from the switch when the HelloReq message with a higher version number is received. The Configuration Distributing function is used to send new configure information to specific switches. The Topology Error Operating function processes the Error message from switches. Also, the Route Responding function is in charge of replying the route inquiry. Figure 4.8 displays the processing flow for all these functions.

The specific processing flow of each function is as follows:

- Feature Updating function

  This function module keeps listening to the incoming HelloReq packets. When the HelloReq packet is received, it will be replied with the HelloRes packet, and its version number will be compared with the previous record in the Node Prefix Table(NPT). If the the version numver is higher than previous record, the NTP will be updated, and the FeatureReq packet will be sent to the switch to get the newest feature data. After that, the controller's Feature Databases(FDB) will be updated.

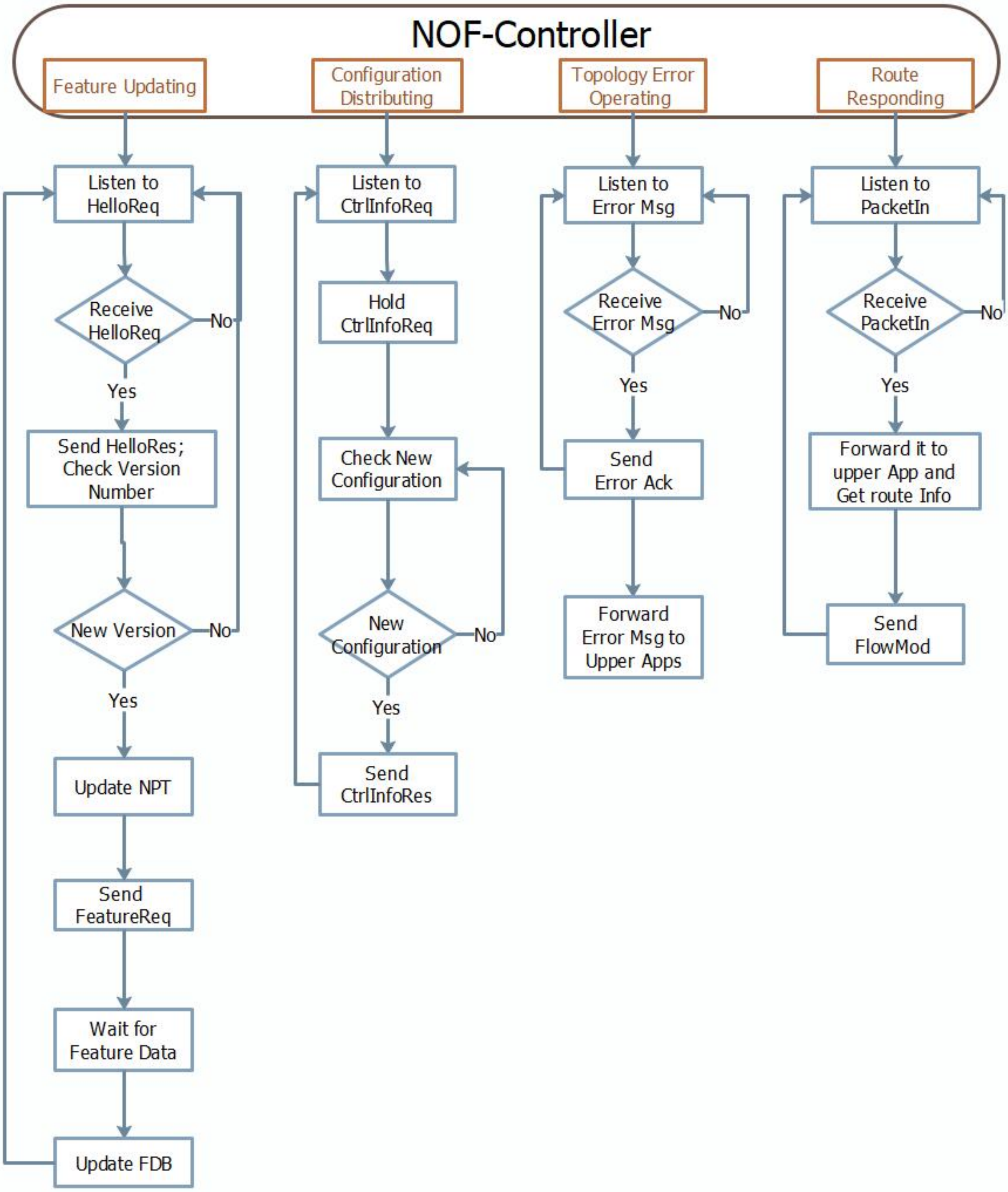


Figure 4.8: NOF-Controller Functions Flowchart

- Configuration Distributing function
  The Configuration Distributing function module receives the CtrlInfoReq messages from all switches and holds them. At the same time, it checks whether some new configuration information needs to be sent to some designated switches. If so, it sends the CtrlInfoRes messages with the configure information to those switches.

- Topology Error Operating function
  This function module listens to the incoming topology Error messages. It is responsible for acknowledging the Error messages and forwarding them to upper applications.

- Route Responding function
  The Route Responding function module is used to process routing queries. It listens for incoming PacketIn packets and forwards them to Upper Apps to get routing decision information. Then it responds the PacketIn message with the FlowMode message, which includes the routing information.

4.3 NDN-based OpenFlow Message Design

This section details the message design for this NDN-based OpenFlow (NOF) protocol. These messages are NDN packets, which are either Interest packet type or Data packet type. Since these messages are relatively small packets, they are not fragmented during the delivery. The other part of this section describes the message interaction between the controller and the switch.

In most cases, controllers are well known, and switches distributed in different locations may not be known in advance. Therefore, in our design, the communication between the switch and the control is always initiated by the switch. The controller advertises a fix name prefix, which is used to let other switches fetch control information. If a switch wants the controller to grab its configuration, status, or feature information, it also needs to advertise a corresponding name prefix to the controller. In our model, we define the controller name prefix that the controller advertised to the whole network:

$$/ndn/ie/tcd/controller01/ofndn/$$

We define the switch name prefix that the switch tells the controller as follows, where {node-id} represents the switch ID:

$$/ndn/\{node-id\}-site/\{node-id\}/ofndn/$$

Referring to the traditional OpenFlow protocol, the header of the OpenFlow packet contains Version, Type, length, and Xid[51]. For consistency and compatibility, the NOF packet we designed also contains these fields. The version is set to N1.0; Type indicates the message type, and its value ranges from 0 to 36. The Length and Xid fields are reserved as 0. In brief, the NOF message header format of this design contains the following information:

$$--/N1.0/[0-36]/0/0/--$$

This header information is appended to the message's name prefix. For example, the name prefix for a message of type 10 sent to the controller is:

$$/ndn/ie/tcd/controller01/ofndn/--/N1.0/10/0/0/--$$

In this NDN-based OpenFlow, all messages' name prefixes follow the format above.

As mentioned in section 3.3, due to the in-network storage mechanism of NDN, the Interest packet transferred between the controller and the switch may be satisfied by the intermediate nodes in the NDN, then it cannot reach the final destination to get the latest version of the information. Solutions to this and other problems are described in Section 4.4. Here we directly assume that the message can be transferred directly to the destination without being satisfied by the intermediate node.

4.3.1 NOF Message Specification

In total, there are 11 types of messages used in this model, and some other types of messages are reserved for future work. They are depicted as follows.

- **Hello Request Message — HelloReq**
  Type Number: 0
  NDN Packet Type: Interest
  Direction: Switch to Controller
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--/\{node-id\}/SN--/ndn/\{node-id\}-site/\{node-id\}/feature$
  Content: Switch-ID, Status Version Number, Switch name prefix
  Description: The switch uses HelloReq message to tell the controller about its status version number(SN) and name prefix, which are used by the controller to check status updates and fetch the up-to-date feature information.

- **Hello Response Message — HelloRes**
  Type Number: 0
  NDN Packet Type: Data
  Direction: Controller to Switch
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--/\{node-id\}/SN--/ndn/\{node-id\}-site/\{node-id\}/feature$
  Content: None/Error
  Description: The controller responds the HelloReq message to the switch with a HelloRes message, whose content is "done", normally. If an error occurs, such as version number mismatched, the message content will be "Error" , including the error code.

- **Error Message**
  Type Number: 1
  NDN Packet Type: Interest
  Direction: Switch to Controller
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/1/0/0/--node-id--type--code--errordetail$
  Content: Error information
  Description: The switch use this message to report errors to the controller. the error type code and details refer to the tradition OpenFlow Error message[52].

- **ErrorACK Message**

Type Number: 1
NDN Packet Type: Data
Direction: Controller to Switch
Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/1/0/0/--node-id--type--code--errordetail$
Content: errorAck
Description: This message used by the controller to confirm the reception of Error message.

- **CtrlInfo Request Message — CtrlInfoReq**
  Type Number: 36
  NDN Packet Type: Interest
  Direction: Switch to Controller
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--/\{node-id\}/SN$
  Content: Switch-ID, Version Number
  Description: The switch sends CtrlInfoReq message to the controller to get the newest configuration. This Interest packet has a long lifetime so that it can be held as long as a new configuration message needs to be sent to this switch. After this interest packet has been satisfied, the switch will send a new CtrlInfoReq message with a higher SN immediately. That means, the controller always holds a CtrlInfoReq message for each switch, and it can reply this message at any time.

- **CtrlInfo Response Message — CtrlInfoRes**
  Type Number: 36
  NDN Packet Type: Data
  Direction: Controller to Switch
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--/\{node-id\}/SN$
  Content: Configuration Information
  Description: The controller uses this message to respond the switch's CtrlInfoReq message if new configuration needs to deliver. Its content includes various actions and parameters. Table 4.5 shows the content of this message.

Table 4.5: CtrlInfoRes Message Content

$Name$	$Type$	$Action$	$Code$	$Parameter$
$FlowMod$	$0x0000$	Add	$0x0000$	$Flow\ elements$
		$Modify$	$0x0001$	
		$ModifyStrict$	$0x0002$	
		$Delete$	$0x0003$	
		$Deletestrict$	$0x0004$	
$FaceMod$	$0x0001$	$Create$	$0x0000$	$Faceid, uri, .$
		$Switchon$	$0x0001$	
		$Destroy$	$0x0002$	$faceid$
		$Switchoff$	$0x0003$	
$TableMod$	$0x0002$	$Sendpacketin$	$0x0000$	
		$flood$	$0x0001$	
		$Drop$	$0x0002$	
$Role$	$0x0003$	$inquire$	$0x0000$	
		$Change$	$0x0001$	
$Error$	$0x0004$	$VersionError$	$0x0000$	
		$AuthenError$	$0x0001$	
		$NodeidError$	$0x0002$	
		$FeatureError$	$0x0003$	
CS	$0x0005$	$Config$	$0x0000$	$Size$
		$Erase$	$0x0001$	
$PrefixMod$		$Advertise$	$0x0000$	$Prefix$
		$withdraw$	$0x0001$	
$FIBrouteMod$		Add	$0x0000$	$route$
		$remove$	$0x0001$	

- **Feature Request Message — FeatureReq**
  Type Number: 5
  NDN Packet Type: Interest
  Direction: Controller to Switch
  Prefix Format: $//ndn/\{node-id\}-site/\{node-id\}/ofndn/feature--/n1.0/5/0/0/$
  Content: None
  Description: The controller uses FeatureReq message to fetch feature information from the switch which has reported the status updated. This message

is sent only if the HelloReq message received by the controller contains a higher version number. Its name prefix is created according to the HelloReq message the switch sent.

- **Feature Response Message — FeatureRes**
  Type Number: 5
  NDN Packet Type: Data
  Direction: Switch to Controller
  Prefix Format: $//ndn/\{node-id\}-site/\{node-id\}/ofndn/feature--/n1.0/5/0/0/$
  Content: Feature Information
  Description: This message is used by the switch to respond the FeatureReq message from the controller with its feature information. Its prefix is the same as the FeatureReq message, and its content is the switch's current feature information.

- **PacketIn Message**
  Type Number: 10
  NDN Packet Type: Interest
  Direction: Switch to Controller
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/node-id--prefix$
  Content: Switch-ID,Querying Prefix
  Description: When the switch cannot route an Interest packet, it sends this message to the controller to query the next hop. This message contains the name prefix of the Interest packet, and it will be responded with FlowMod message.

- **FlowMod Message**
  Type Number: 10
  NDN Packet Type: Data
  Direction: Controller to Switch
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/node-id--prefix$
  Content: Route Information, FlowEntry elements

Description: The message used by the controller to respond the PacketIn message from the switch. The controller uses it to adjust the FlowEntry in the switch. Its content includes all the elements of the FlowEntry and the actions used to operate the FlowEntry. The details of the content are described in Table 4.6.

Table 4.6: FlowMod Message Content

$Field$	$Value$
$Match$	$EP, Face, Prefix, Priority$
$Cookie$	$None$
$Command$	$Add(0x0000)/Modify(0x0001)/ModifyStrict(0x0002)/$ $Delete(0x0003)/Deletestrict(0x0004)$
$Idle\_timeout$	3600
$Hard\_timeout$	36000
$Priority$	1
$Buffer\_ID$	$*$
Out_port	$Face-ID$
$Flag$	$None(0x0000)/SendFlowRemoved(0x0001)/$ $CheckOverlap(0x0002)/Emerg(0x0003)$
$Action$	$Forward(0x0000)/Flood(0x0001)/Enqueue(0x0002)/$ $Modify(0x0003)/Drop(0x0004)$

- **FowRemoved Message**
  Type Number: 11
  NDN Packet Type: Interest
  Direction: Switch to Controller
  Prefix Format: $/ndn/ie/tcd/controller01/ofndn/--/n1.0/11/0/0/--/node-id--prefix$
  Content: Removed Prefix
  Description: When a FlowEntry expired, and its flag field is set as "Send FlowRemoved", the switch uses this message to report to the controller. This message is an Interest packet, but it does not need Data packet to satisfy.

Some other types of messages are reserved for future work. They are listed in Table 4.7 briefly.

Table 4.7: Reserved Message Type

Msg Name	*TypeNo.*	*Direction*	*Description*
GetConfigReq	7	*Controller to Switch*	*Msg Fragmentation Configuration*
GetConfigRes	7	*Switch to controller*	
SetConfig	9	*Controller to Switch*	
FaceMod	16	*Controller to Switch*	*Control Faces Status*
MultipartReq	18	*Controller to Switch*	*Request individual flow information*
MultipartRes	18	*Switch to controller*	
BarrierReq	20	*Controller to Switch*	*Set a sychronization point*
BarrierRes	20	*Switch to controller*	
RoleReq	24	*Controller to Switch*	*Change controller's role*
RoleRes	24	*Switch to controller*	

4.3.2 Interaction between Switch and Controller

This section describes how the controller and the switch interact using the messages. Figure 4.9 shows the interaction process.

In this picture, it needs to be clear that the processes of some messages are executed simultaneously. The switches and controller are designed to work in multiple threads so that they can handle multiple tasks concurrently. It is necessary for our model. For instance, while waiting for the latest configuration information, the switch may also need to send PacketIn message to inquire routing information. Of course, some messages are processed in-order. For example, the controller only sends FeatureReq packets after receiving a HelloRep packet with a higher version number. The details are explained as follows:

Under normal circumstances, the controller starts up and advertises a fixed name prefix, and then enters the listening status, waiting for the incoming NOF messages. When the switch starts, it first checks for changes in its status, such as FIB or Faces changes. Since it is the first time to check them, many changes are detected because the faces and channels just startup. The switch then sends a HelloReq message to the controller, which contains a higher version number and its advertised prefix. This prefix is advertised when it sends the HelloReq message. When the controller receives this HelloReq message, it responds the message with HelloRes and checks whether

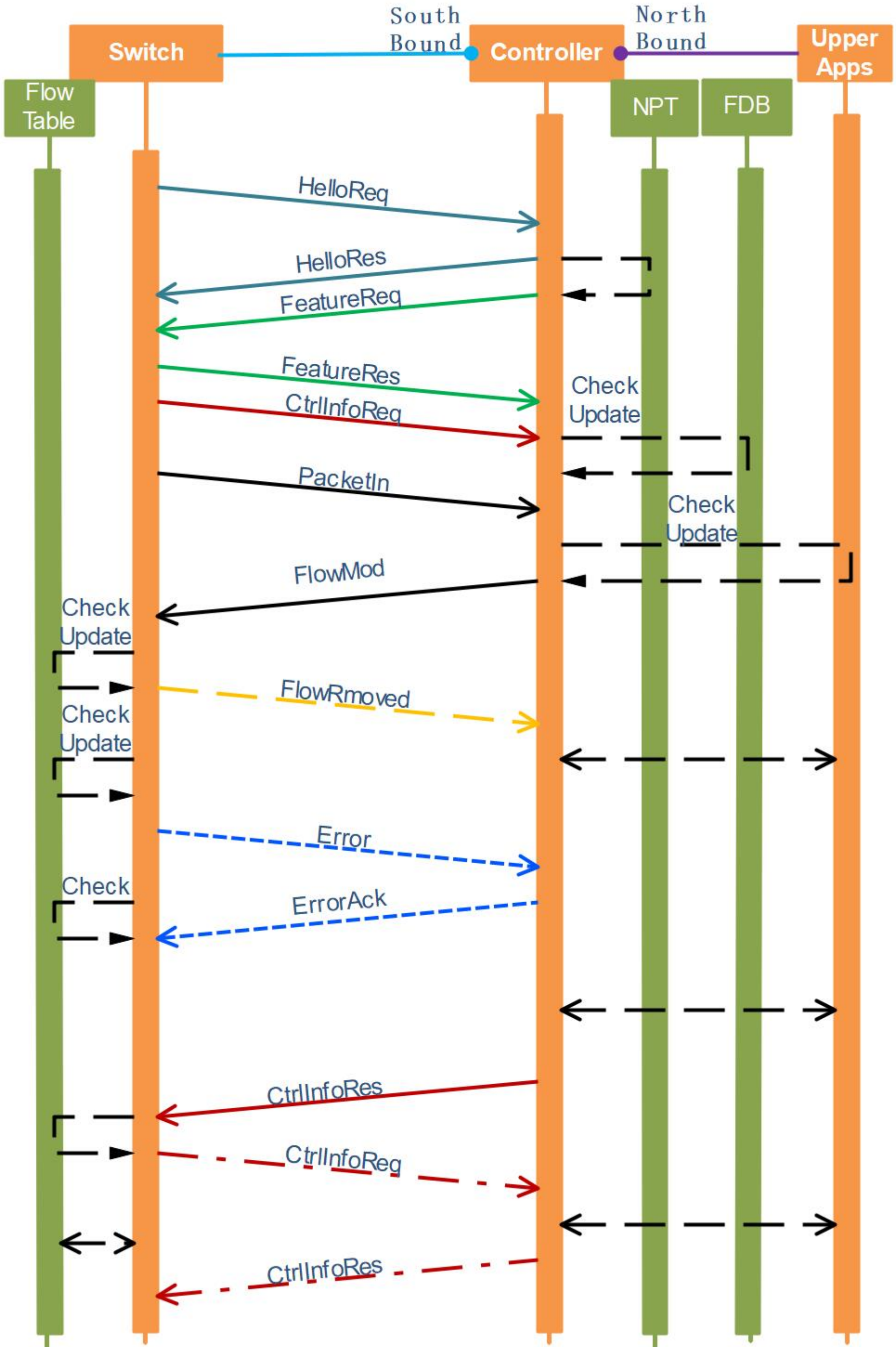


Figure 4.9: NOF Message Interaction

it is the updated HelloReq message by comparing with its NPT. If so, the controller immediately updates its NPT and sends a FeatureReq message to the switch to fetch the up-to-date Feature information. The switch uploads its up-to-date feature information within the FeatureRes message. Finally, the controller receives the FeatureRes message and updates its Feature Databases. The switch and controller use a thread to do the above work, respectively.

After booting up, the switch sends a CtrlInfoReq message to the controller. This message is an NDN Interest packet, which has a long lifetime. Its purpose is to let the controller reply to the switch when there is updated configuration information. The controller receives and holds it until a piece of configuration information needs to be transferred to the switch. When the switch receives the CtrlInfoRes message, it sends a higher version number of CtrlInfoReq message to the controller, so that it can always get the newest configuration information from the controller immediately.

The switch has a process for monitoring the notification of the NDN Node. When it detects that there is an Interest packet that cannot be forwarded since there is no next-hop for the prefix of the packet in the FIB and there is no corresponding FlowEntry in the FlowTable. The switch sends a PacketIn message to the controller to ask how the packet is handled. When the controller receives the PacketIn message, it responds with a FlowMod message, which can add or modify the FlowTable of the switch to instruct the switch how to deal with the previously queried packets.

When the switch needs to report an error to the controller, such as topology changes, it can send an Error message directly with an error code. The controller confirms that the Error message has been received by replying an ErrorAck message. If this error causes a configuration change, the controller will alter the configuration of corresponding switches through the CtrlInfoRes message.

When a FlowEntry expires, and its “Flag” field is set to the value “0x0000”. Then the switch sends a FlowRmovd message to notify the controller.

4.4 Solution of Challenges

This section gives the solutions to challenges discussed in section 4.4.

1. How to carry data in the Interest packet.

There is no “Content” field in the NDN Interest packet structure, so by default, it cannot carry any payload. However, in our model, we need some Interest packets to take some extra information. For example, the PacketIn message is an Interest packet that needs to carry an inquiring name prefix. There are two proposals to address this issue. The first one is to use the “ApplicationParameters” field in the Interest packet. This field is designed to transfer parameters for applications, which is ideal for use in our model. But the functionality of this field is still under development and will not be available for the time being. Another alternative is adding payload directly to the end of the Interest packet’s name. For example, the Interest packet’s name is “$/ndn/abc/def$” and the payload is “$/mn/xyz/$”we can combine them as “$/ndn/abc/def/--/mn/xyz/$” and this Interest can be caught in the destination which is listening to the prefix “$/ndn/abc/def/$”. so It will not affect the packet delivery. There are two advantages to use this method. first, it does not modify the structure of the Interest packet, nor does it require any adjustments to the existing NDN architecture. In addition, this method can be easily migrated to the former method. When the “ApplicationParameters” field is available, adding payload directly to this field is available, and no further adjustments are required.

2. How to get the up-to-date information from the controller.
   Since the in-network storage mechanism in NDN, the Interest packet are intended to to be satisfied by the closest NDN node. However, in our model, the switch needs to get the up-to-date information directly from the controller instead of getting outdated data from the intermediate nodes. The solution to this problem is to set the value of the “setFreshnessPeriod” parameter in the “MetaInfo” field in the Data packet to “ 0 ”, which means the Data will be marked as “not Fresh” immediately after it is stored in a node’s CS. Meanwhile, the “MustBeFresh” field in the Interest packet is set as “True” so that the Data in the CS of all intermediate nodes cannot satisfy the Interest packet, and the Interest packet will be transmitted to the data source, the controller, to be satisfied.

3. How to distinguish different types of messages.
   The controller advertise a fixed name prefix for the Interest packet accessing. This prefix used in all kinds of NOF messages, so the problem is that,how to

recognize them. In our model, the NOF header, which includes the protocol version, type, length, and xid, is added into the name of the Interest packet. See section 4.3. So different messages can be distinguished according to their types, even they have the same Interest name prefix. The data producer uses an Interest filter to select different messages.

4. How the controller proactively sends messages to a specific switch.
   In our model, the controller is unable to actively send information to a switch because switches usually do not pre-advertise a name prefix for accessing. They only advertise a prefix for uploading feature data. The question then is how the controller send commands or data to a specific switch as needed. The solution is that, the switch first sends a CtrlInfoReq message to the controller, which has a quite long lifetime. The controller holds this message until a new configuration needs to be sent to this switch. Then the controller uses CtrlInfoRes message, which includes the commands or configuration, to satisfy the previous CtrlInfoReq message. In this way, the controller can send data to any switches at any time.

5. How the NFD invokes the NOF switch agent's functions.
   To integrate our model into the original NDN is a big challenge. Adjusting the original NDN architecture and underlying protocols is unacceptable. We use the NFD Management protocol[53], which allows users, tools, and other programs to retrieve, monitor, and modify NFD forwarder status. The NFD node advertises a name prefix, and other programs can use this name prefix to fetch the NFD information and logs or check its status. The switch agent in our model listens to these prefixes and use the events to trigger the function modules.

4.5 Summary

This chapter mainly introduces the prototype design, which consists of four sections. The architectural design is described first, including the switches and controller structures. The approach of integrating the NDN node and the Switch Agent is expressed in the switch architecture design; the controller's elements and their functions are depicted in the controller architecture design. Then, the subsequent section details all the message types used in this model, as well as their formats and usage. How the controller and switch interact with each other using these messages is also outlined in this section. In the end, the solutions to the main challenges are explained in detail.

In conclusion, this chapter provides an in-depth description of the design details of NOF SDN architecture. The next chapter will introduce the implementation of this model.

Chapter 5

Implementation

This chapter displays the implementation of the NDN-based OpenFlow protocol architecture. The demands of the experimental environment are listed in Section 1. The second section describes the details of the implementation, which includes the switch part and the controller part. Their specific implementation methods and key code are presented. The next section shows how the NDN packet processed in our model, followed by the source code and its running method. A brief overview is given at the end .

5.1 Implementation Environment

This section describes the implementation environment, including the system environment, development tools, simulators, and implementation topology.

5.1.1 System Environment

The model in this project is based on the Linux system. The system we used is Ubuntu 18.04.2 LTS, which can be downloaded from "*https://ubuntu.com/download/desktop* ". The system runs on the VMware Workstation Player 15 tool, whose free version can be found on "*https://www.vmware.com/*". Since we need to install the necessary tools in this system, the administrative privilege is required. Sufficient hard disk space is also needed because our NDN simulator is also running on the system.

5.1.2 NDN Simulator

The purpose of this project is to design a new OpenFlow protocol that can run on the NDN. So we need an NDN environment to execute our model. Mini-NDN is a lightweight NDN platform simulator based on Mini-CCNx, which is a branch of Mininet. It is an open and free software which uses the NDN libraries, NFD, NLSR, and NDN tools for experimentation, research, and testing NDN projects[54].

The underlying code of Mini-NDN is written in C++, and its latest version is Mini-NDN 0.4.0, which is based on NFD version 0.6.5[55]. About Mini-NDN installation and basic operation are presented on [56].

5.1.3 Development and Test Tools

- PyNDN
  NDN Common Client Libraries(NDN-CCL) is an API for for user applications to use NDN, which are written in several languages, such as Java, JavaScript, Python, C++, and .NET. In our project, the model is developed in Python. So the Python version of NDN-CCL is required, which is called PyNDN. The latest version of PyNDN is PyNDN2, and it can be downloaded from "*https://github.com/named-data/PyNDN2*", and its documentation can be found in [57].

- PyCharm
  PyCharm is a Python IDE with a set of tools that help users improve their productivity when developing in Python, such as debugging, syntax highlighting, Project management, code jumping, smart prompting, auto-completion, unit testing, version control. PyCharm can be download from "*https://www.jetbrains.com/pycharm/*". Pycharm has a free trial and a professional version. Because our model runs on a remote NDN simulator, its "SSH Interpreter" feature is needed, so we use the Professional Edition, PyCharm 2019.1 (Professional Edition).

- Git/Github
  Git is an open-source distributed version control system that efficiently and quickly handles project management from very small to very large. The code

can be backed up to the Github online repository. Version control is done with Git/Github in our project. Git can be downloaded directly by command:
sudo apt-get install git.

- Wireshark
  Wireshark is a network packet analysis software, which is used to capture network packets and display the most detailed network packet data as much as possible. Wireshark is designed to capture and analyze IP packets. In our NDN environment, although the underlying channels are used IP/UDP, the Wireshark still cannot recognize the contents of the NDN package. Therefore, a plugin that can parse NDN package is needed. Wireshark can be downloaded from "*https://www.wireshark.org/*", and the download and usage of the NDN plugin can be found in [58].

- Xserver Xserver is also called X11 or X Window System, which provides the basic framework for a GUI environment for Unix-like Operation System. We use Xserver to display the graphical interface of the remote Linux system and perform multiple interfaces operations.It contains both server and client programs. The client should be installed on local windows system, which can be download from "*https://www.x.org/wiki/XServer/*". Its server should be installed on remote linux system with the command:
  sudo apt-get install xorg.

- Other NDN testing tools
  Some of the testing tools that come with NDN are also useful, such as NDN-ping. NDN-dump, NDN-peek, NDN-chunks, AND NDN-dissect. These tools can be optionally installed when installing the NDN simulator. In our projects, we use them for network testing.

5.1.4 Implementation Topology

Figure 5.1 is the basic implementation topology for this project. This topology holds the network structure required in our model implementation, which includes two switches, one controller, one producer, and a consumer. These nodes are all simulated by Mini-NDN, and run on the NDN. By default, they can transfer data through the NDN and benefit from all the features of the NDN, such as in-network storage, Interest aggregation, and connectionless transmission. In this topology, one switch and controller have physical line connections, and the other switch does not. The communication between the switches and the controller is through NDN. This topology is similar to the real-world application scenario, which can be extended to any more complex NDN environment in theory.

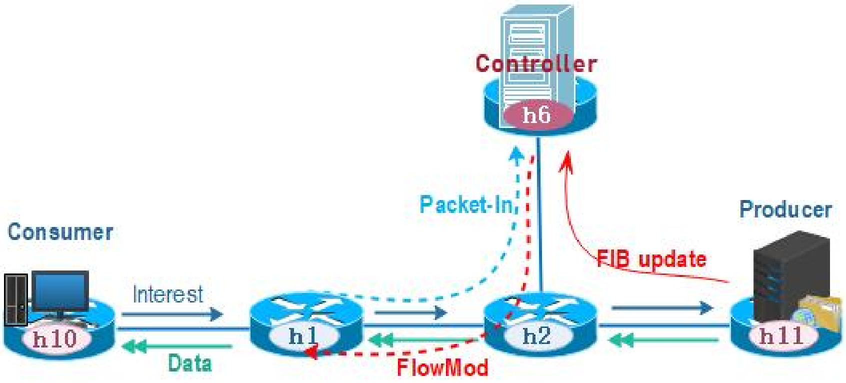


Figure 5.1: Implementation Topology

5.2 Implementation Approach

Section 4.2 describes the design of the various functional modules of the switch and controller, as well as their processing flow. This section details the specific implementation of these functional modules.

5.2.1 Implementation of Switch Functions

In this model, the switch uses a multi-threaded architecture so that it can handle multiple tasks simultaneously. As shown in Figure 5.2, the switch has five main threads, which operate five different functional modules.

- Status Monitoring Module
  This module periodically checks the status changes of the switch. By default, it checks once per second. The items it checks include Channels, Faces, FIB, and

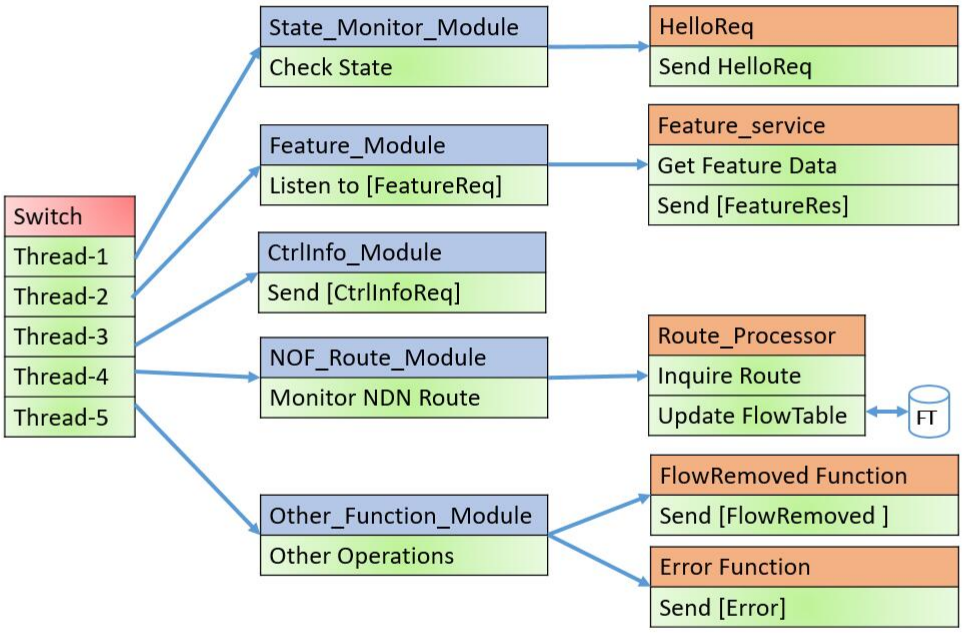


Figure 5.2: Switch Software Structure

RIB. If changes happen, the "HelloReq" Class will be called to send a HelloReq message with a higher version number to the controller. The module uses the NFD management protocol[53] to monitor the switch status. That is, it sends a specified prefix to the NDN to obtain the current status information, then compares with the previous record.

Besides, it sets a threshold to prevent specific faces from becoming unstable and link flapping. If an item changes more than five times in 5 seconds, then the status will be held for 300 seconds, during the time the status will not be checked.

Key Code:

```
        while True:
            if self.status_update_checker():
                HelloReq().run(self.hello_version_number)
            time.sleep(1)

            # protect flapping case
            if (self.channels_flapping_time > 5 or
            self.faces_flapping_time > 5 or
            self.fib_flapping_time >5 or
            self.rib_flapping_time >5):
        time.sleep(300) # become silent for 5 min

        self.channels_flapping_time = 0
        self.faces_flapping_time = 0
        self.fib_flapping_time = 0
        self.rib_flapping_time = 0
```

- Feature Module

  This module uses a thread to listen for FeatureReq messages sent from the controller. After receiving the message, the relevant function is called to obtain the current features of the switch, that is, to obtain the FIB and face information. The obtained feature information is then sent to the controller with a FeatureRes message.

  Key Code:

```
def onInterest_Feature(self, mainPrefix, interest, transport,
    registeredPrefixId):
    print("++++++ Received <<<FeatureReq>>> interest ++++++ \n")
    feature_face = OSCommand.getface() #get face info
    feature_FIB = OSCommand.getFIB() #get FIB info
    nodeid = bytes(self.nodeid,'utf-8')

    #combine the features:
    feature_data =nodeid + b'----' + feature_face + b'----' +
        feature_FIB

    #encapsulate feature to Data packet
    data = OFMSG().create_feature_res_data(interest,feature_data)

    transport.send(data.wireEncode().toBuffer())
    print("++++++++ Sent <<<FeatureRes>>> Data ++++++++ \n")
```

```
        self.isDone = True #mark the Feature Interest has been
            satisfied.
```

- CtrlInfo Module

  This module is used to get configuration information from the controller. It sends a "configuration request" Interest packet to the controller, which called "CtrlInfoReq" message. The lifetime of this Interest package is very long, so it will not time out. This message will only be responded to when the controller has a new configuration to send to the switch. Otherwise, the Interest will be held. After receiving the configuration data sent back by the controller, this module will send a new CtrlInfoReq message with a higher version number immediately so that the controller can send configuration data at any time.

  Key Code:

```
ctrlinfo_version_number = 100001 #Initial version number
while True:
    #if the last Interes has been satisfied
    if (self.send_ctrlinfo_interest):
        try:
            self._sendCtrlInfoInterest(ctrlinfo_version_number)
            ctrlinfo_version_number += 1
            self.send_ctrlinfo_interest = False # repeat to send this
                interest.
            self.isDone = False
            ......
```

- NOF Route Module

  This module is designed to monitor the forwarding status of local NFD. When an Interest cannot be forwarded because of the lack of next-hop, this module will call the "OF_Route_Processor" class, which check the local FlowTable first, if there is no corresponding FlowEntry, it will call the PacketIn class to inquire the route and update its FlowTable.

  Key Code:

```
    #Check NDN FlowTable first, if no result,...
    if (not NdnFlowTable.searchitem(prefix)):
        PacketIn().run(prefix) #send PacketIn including the prefix.
            ......
```

- Other Function Module

  This module is used to handle some other functions. For example, when the FlowEntry expires, it may need to send a FlowRemoved message to the controller; or when the switch detects an error, it will send an Error message.

5.2.2 Implementation of Controller Functions

Considering that the controller may manage many switches, its workload could be very heavy, so the controller in this project uses a multi-process design, instead of multi-thread.

Figure 5.3 show the controller’s software structure, which includes four processes, and each of them operates different functional modules.

- Hello Function Module

  This module listens for incoming HelloReq messages. When receiving the HelloReq message, the controller checks its version number and compares it with the version number of the corresponding entry stored in the NPT. If the version number of the received message is higher, the controller will update its NPT item. Meanwhile, it calls the ”Feature” class, which can send FeatureReq message to fetch the up-to-date feature information of the switch. After receiving the feature information, this module will use it to update its Feature Database.

  Key Code:

```
def updatenodeprefixtable(originalinterest):
    helloreq_name_prefix =
        NodePrefixTable.parse_interest(originalinterest)
    featurereq_interest_prefix = str(helloreq_name_prefix[3]) +
        ’/--/n1.0/0/0/0/’
    #NPT Format:[h1, version number, type_number, node_prefix]
```

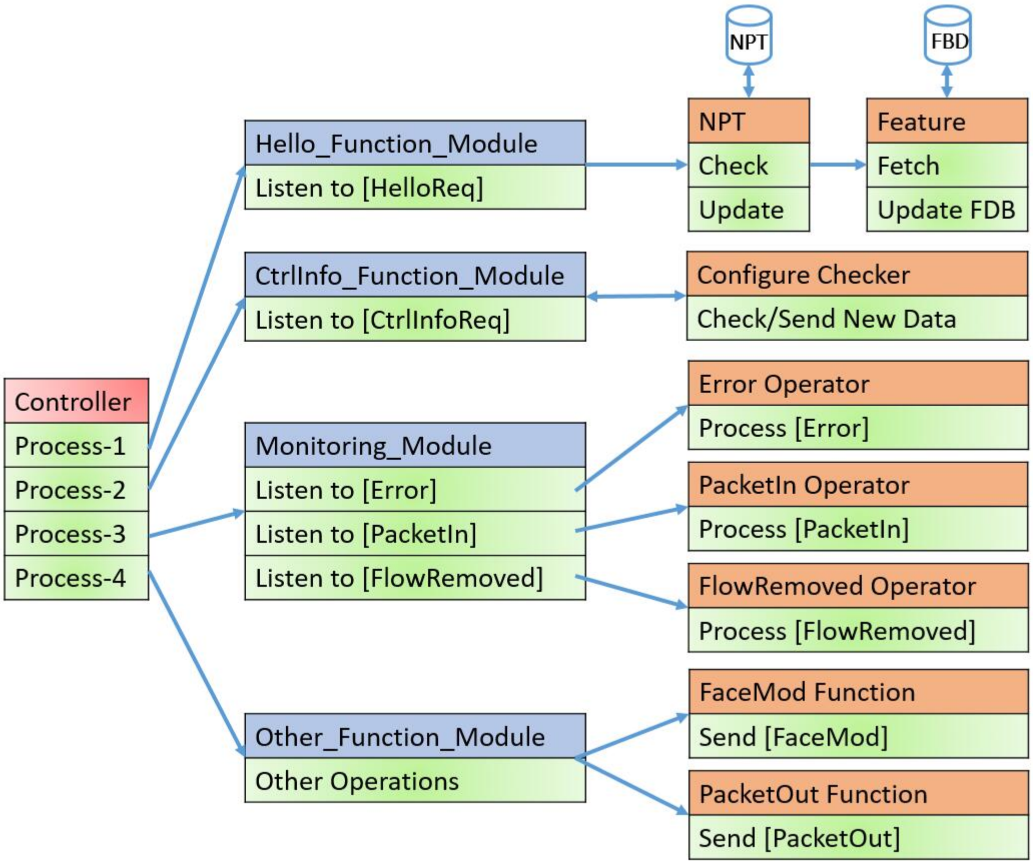


Figure 5.3: Controller Software Structure

```
if helloreq_name_prefix[0] in NodePrefixTable.NPT[:, 0:1]: #check
    existed item
    nodeindex = NodePrefixTable.searchitem(helloreq_name_prefix[0])
    if helloreq_name_prefix[1] >
        int(NodePrefixTable.NPT[nodeindex][1]):
        #higher version number message----update NPT
        NodePrefixTable.NPT =
            NodePrefixTable.delitem(NodePrefixTable.NPT, nodeindex)
        NodePrefixTable.NPT =
            NodePrefixTable.additem(NodePrefixTable.NPT,
            helloreq_name_prefix)
        try:
            #call Feature class to fetch feature data
            FeatureReq().run(featurereq_interest_prefix)
            print('feature request has been sent out ')
        except:
            print("FeatureReq send out fail")
    else:
        print("the same hello message received") #same version
            number
else:
    FeatureReq().run(featurereq_interest_prefix) #No existed item
        in NPT.
    print('feature request has been sent out ')
    NodePrefixTable.NPT =
        NodePrefixTable.additem(NodePrefixTable.NPT,
        helloreq_name_prefix)
......
```

- CtrlInfo Function Module

  This module is designed to detect incoming CtrlinfoReq messages. When the controller receives the CtrlinfoReq messages from all switches, it will hold them first. Then the configure checker is called to check for new configuration information. If new configuration information needs to be sent to the specified switches, the CtrlInfo Function Moule encapsulates the configuration information as data into the CtrlInfoRes message and uses them to respond to the specific switches. In fact, it uses the Data packet to satisfy the previously received Interest packet.

Key Code:

```
#Loop check new configure info.
while (self.new_CtrlInfo_data == self.CtrlInfo_data):
    time.sleep(15)

#get new data
self.CtrlInfo_data = self.new_CtrlInfo_data
data = self.ofmsg.create_ctrlinfo_res_data(interest,
    self.CtrlInfo_data) #encapsulate
transport.send(data.wireEncode().toBuffer()) #send data
    ......
```

- Monitoring Module The Monitoring Module is a listener for listening to other messages. In our model, it can listen for Error, PacketIn, FlowRemoved messages currently, and assign them to different operators for processing. For example, when a PacketIn message is received, it calls the PacketIn Operator for further processing. The PacketIn Operator extracts the prefix information that needs to be queried to the Upper Applications for routing decision. After that, it sends FlowMod message to the switch to adjust the FlowTable.

  Key Code:

```
#filters:
# for Error msg
error_msg_prefix =
    Name('/ndn/ie/tcd/controller01/ofndn/--/n1.0/1/0/0/')
self.face.setInterestFilter(error_msg_prefix, self.onInterest_ErrorMsg)

#for packetin msg
packetin_msg_prefix =
    Name('/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/')
```

```
self.face.setInterestFilter \
(packetin_msg_prefix,self.onInterest_PacketIn)

#for FlowRemoved msg
FlowRemoved_msg_prefix =
   Name('/ndn/ie/tcd/controller01/ofndn/--/n1.0/11/0/0/')
self.face.setInterestFilter \
(FlowRemoved_msg_prefix,self.onInterest_FlowRemoved)

   ......
```

- Other Function Module
  This module handles other the controller's operations. For example, send a FaceMod message, or test to add other functions, such as sending a PacketOut message. This module is not a critical module in our model. It is designed to add more functionality in the future.

5.3 NDN Packet Processing Sequence

This section illustrates the processing flow of NDN packets in our design model. The implementation topology is still used since it can represent the general NDN network structure. It is assumed that the Producer offers data named "*/ndn/h11/dissertation/v1*" and it has registered this prefix to switch "h2". The Consumer fetches this data by sending an Interest packet with the same name. Both the switch "h1" and switch "h2" do not have the route(next hop) to forward the Interest packet to the data source, the Producer, so they have to ask the controller for route information.The process sequence is displayed in Figure 5.4. In this case, we don't care about the communication process between the controller and the upper apps, since they are out of the scope of this dissertation. We just assume that they can provide the route information for switches.

- Step 1: The Consumer create an Interest packet, which is a standard NDN packet with a name of "*/ndn/h11/dissertation/v1*".

- Step 2: The Consumer sends the Interest packet to switch "h1".

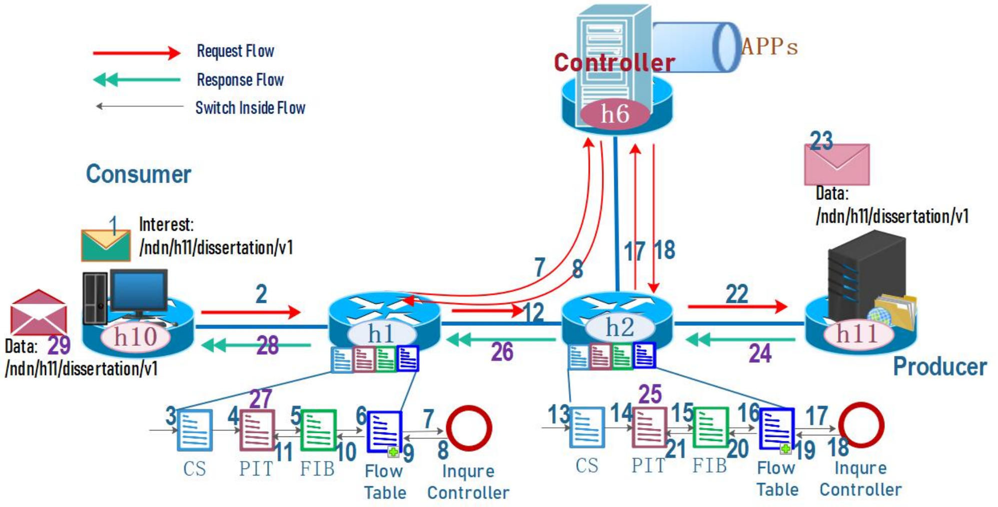


Figure 5.4: NDN Packet Processing Sequence

- Step 3: Switch “h1” searches its ContentStore using the Interest packet’s name prefix to find data which can satisfy this Interest. In our case, it is the first Interest for the name prefix, so there is no matched data in the CS OF switch “h1”.

- Step 4: Switch “h1” checks its PIT to confirm if the Interest with the same name prefix has already been forwarded. In our case, there is not.

- Step 5: Switch “h1” searches its FIB to find the next-hops to forward the Interest packet. But there is no FIB item matching the Interest’s name prefix in this case.

- Step 6: The NFD “no-route” notification in switch “h1” triggers the switch agent in our model, which first checks the local FlowTable, but no matched FlowEntry found.

- Step 7: The switch agent in switch “h1” sends a PacketIn message to the controller. This PacketIn message is an NDN Interest packet, which includes the prefix used for route inquiry, and it can be delivered to the controller directly via NDN.

- Step 8: The controller uses FlowMod message to respond to the PacketIn message, which includes the route information of the inquired prefix. The FlowMod message is an NDN Data packet, which can be sent to the switch directly.

- Step 9: The switch agent adds the FlowEntry to its FlowTable according to the received route information.
- Step 10: The switch agent adds FIB item for this prefix.
- Step 11: The switch "h1" adds a record of the original Interest packet to its PIT.
- Step 12: Switch "h1" forwards the original Interest packet to switch "h2".
- Step 13 - 22: The steps 13-22 done on switch "h2" is the same as the steps 3 to 12 done on switch "h1".
- Step 23: The Producer receives the Interest packet and encapsulates corresponding data to a Data packet, whose name is the same as the original Interest packet.
- Step 24: The Producer sends the Data packet to Switch "h2".
- Step 25: Switch "h2" checks its PIT and finds out the pending Interest record.
- Step 26: Switch "h2" forwards the Data packet to switch "h1" according to the PIT record, then deletes the PIT record.
- Step 27 - 28: Switch "h1" forwards the Data packet the same as switch "h2" have done in steps 25 -26.
- Step 29: The Consumer finally receives the data it interested.

5.4 Source Code Implementation

The source code for this project has been published in the GitHub repository:
https://github.com/lijian2020/NDNAPP.git.

To run the source code, the NDN environment is required. The deployment of Mini-NDN is described in [56]. In addition, the Python API, PyNDN2, is also necessary, and its instruction is outlined in section 5.1.3.

After preparing the experimental environment, download the source code to the home folder. First replace the /home/mininet/mini-ndn/NFD/daemon/fw/best-route-strategy2.cpp file in the system with /doc/best-route-strategy2.cpp. Then recompile and install the NDN environment with the following command:

```
$ cd /home/mininet/mini-ndn/NFD
$ sudo ./waf configure
$ sudo ./waf
$ sudo ./waf install
```

The experimental environment is ready now. Start the Mini-NDN with the command:

```
$ sudo minindn
```

Then use the following command to run each component on the NDN node.

Controller:

```
$ cd ./NDNAPP-master
$ sudo python3 controller.py
```

Switch:

```
$ cd ./NDNAPP-master
$ sudo python3 node.py
```

Producer:

```
$ cd ./NDNAPP-master
$ sudo python3 testproducer.py -n /ndn/abc/xyz
```

Consumer:

```
$ cd ./NDNAPP-master
$ sudo python3 testconsumer.py -u /ndn/abc/xyz
```

The name prefix “/ndn/abc/xyz” in the Producer and Consumer’s commands can be any other names. It is used for test.

5.5 Summary

This chapter details the implementation of NDN-based OpenFlow protocol.

It first introduced the environmental requirement, including the system, simulator, tools, and the implementation topology.

The second section focuses on the approaches of the model implementation, which consists of two parts, the switch functions implementation and the controller functions implementation. This section also shows how our model implements multiple functional modules through multi-thread and multi-process. Based on them, the switch implements the Status monitor module, feature module, CtrlInfo module, NOF Route module, and other functions module through 5 threads, while the controller adopts the design of four processes, which are responsible for implementing the Hello Function module, CtrlInfo. Function module, Monitoring Module, and other function modules. Some key code is displayed as well.

The NDN packet processing sequence is also depicted in this chapter. It shows the complete processing steps of the NDN packet under our NDN-base OpenFlow architecture.

Finally, this chapter gives the complete experimental environment requirements, source code, and the method to run the code. The next chapter provides a detailed assessment of the entire design.

Chapter 6

Evaluation

The motivation of this project is to integrate NDN and SDN to combine their advantages. We referred to the OpenFlow protocol and redesigned an SDN protocol that can run directly on the NDN, the NDN-based OpenFlow protocol (NOF), which has similar functions to the traditional OpenFlow protocol: network management and routing decision. Therefore, the evaluation of this project should focus on its functionality. The functions and implementation of this model are described in detail in the previous chapters. This chapter converges on evaluating the performance of these features.

This chapter includes four parts. The first section lists the specific content that needs to be evaluated, including its functionality, scalability, compatibility, and security. Then the second section introduces the evaluation methods for this project, including the tools and specific steps. The results of the assessment and the discussion will be presented in the third section that follows, followed by a conclusion of this chapter.

6.1 Evaluation Contents

This project is to port OpenFlow to NDN, so evaluating the performance of its functions is the focus. It also needs to assess its scalability, security, and compatibility.

In terms of functionality, we need to evaluate the performance of the functions mentioned in the previous sections. It mainly includes the following contents:

1. The functions of the status monitoring and the switch sending status report to

the controller.

2. The controller handles the status report and its ability to retrieve the up-to-date features of the switch.

3. The ability of the switch sending a configuration request and the ability of the controller responding configuration information to the switch.

4. The function of switch inquiring routes and updating its FlowTable.

5. The function of controller handling the route query.

6. Other switch's function and other controller's functions.

The above functions cover network management and routing decision and are all the characteristics of this model design. It basically achieves the primary duties of the traditional OpenFlow protocol, and its modular design makes it easy to add more features.

In terms of scalability, our model can theoretically support a large number of network devices (NDN nodes), but also can be extended to more complex network environments. We need to confirm that the functions of our model are working perfectly in a more complex network.

As for security and compatibility, we need to evaluate the security mechanisms and the scope of compatibility of this model.

6.2 Evaluation Methods

The implementation of our model is done in a simulated environment, so the simulator is needed for the evaluation. In addition, the main functions of the model are accomplished through the NOF messages, so the packet capture tool is also necessary for the model evaluation and analysis. In our case, We use Wireshark because its plugin supports NDN packet analyzing.

In short, the evaluation tools:

- Simulator: Mini-NDN

- Packet capture tool: Wireshark

The project redesigns the OpenFlow protocol to enable the functions of network management and routing decisions on the NDN. These functions are the core of the project, so the evaluation focuses on whether its functions can be achieved as expected. In addition, another objective of our model is that it can be deployed in a large size and complex NDN environment, so its scalability needs to be verified. Furthermore, its compatibility and security are also important factors for usability. Therefore, in our model, the evaluation approaches and metrics are shown below:

- For functionality evaluation, the method we use is to verify its availability. We test all the model's features and use cases, analyze all the packets to confirm if it works as expected.
- For scalability assessment, the strategy we use is to deploy our model and verify its functionality with different network scales. The specific approach is to deploy our model and verify availability in a small, simple network environment, and then implement the model in a relatively large, complex network and prove the feasibility. Finally, we discuss its scalability in theory.
- For the evaluation of security and compatibility, we mainly judge it by analyzing its architecture design.

The detailed evaluation process and results are shown in the next section.

6.3 Evaluation Process and Results

This section is divided into four subsections that provide a comprehensive assessment of the functionality, scalability, compatibility, and security of the NDN-base OpenFlow model.

6.3.1 Functionality Evaluation

Section 4.2 illustrates the distinct functional modules of the switch and the controller and their workflow. The description of how these functional modules work is presented in Section 5.2. This section verifies the procedures and results performed by these modules.

Figure 5.1, the Implementation Topology, is used for the functionality evaluation in this section since it contains the basic elements of an NDN environment: the controller, the directly connected switch, and the indirectly connected switch. It is simple but enough to verify the functionality. Other evaluations in later chapters use different topologies depending on the assessment contents. For these topologies, we assume that the underlying NDN of this model is up and running, and the switches and controller can communicate with each other via NDN packets. The basic communication between NDN nodes is not the focus of this evaluation.

In our model, the implementation of particular functions requires the switch and controller to cooperate with each other. So when verifying certain features, their corresponding features in peers are tested simultaneously.

The detail evaluation process for each functional module is as follows:

- **Function Module 1: Status Monitoring Function in the switch and Feature Updating Function in the controller.**

  Design Principle:

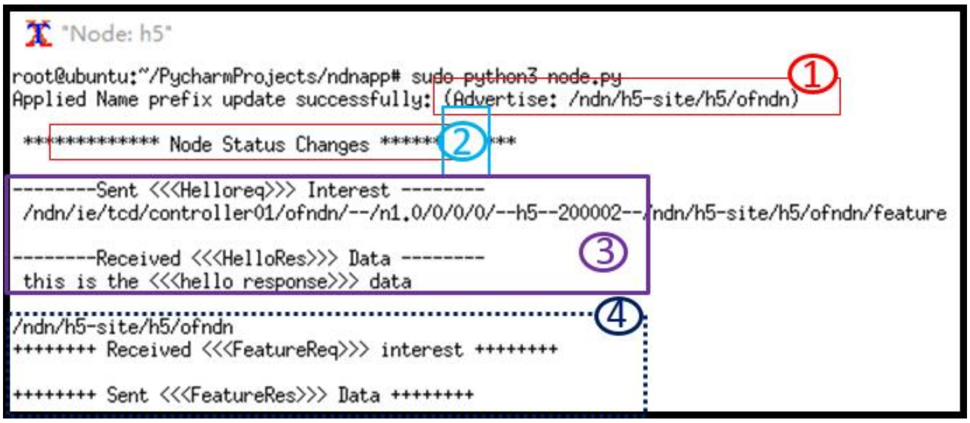
  The process of this function module involves both the switch and the controller. When the changes of status are detected, the switch sends a higher version of the HelloReq message to the controller. After receiving the HelloReq, the controller sends a HelloRes acknowledgment packet and updates the NPT. Meanwhile, it sends a FeatureReq message to the switch to obtain the up-to-date feature information. The switch then responds it with the FeatureRes message. At last, the controller updates its Feature Database based on the content of this message.

  Evaluation Methods:

  In order to test this function module, we start a switch and a controller. After they have finished booting, we modify the switch's features, such as destroy a face, then we inspect the processes in the switch and the controller. Also, we capture and analyze the packets they use to communicate. At the same time, we can also output the database content of the controller to verify whether its function is achieved.

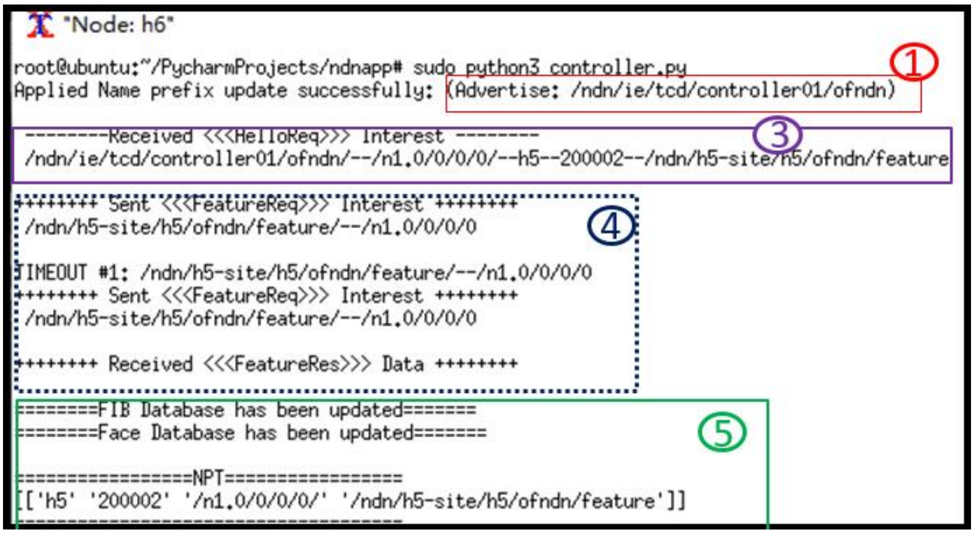
  Evaluation Result:

  Figure 6.1 and Figure 6.2 show the experimental results of the switch and controller, respectively. Figure 6.3 displays the relevant data packets captured via Wireshark. The outputs(NPT, FDB) of the database contents in the controller are present in Figures 6.4.



```
"Node: h5"
root@ubuntu:~/PycharmProjects/ndnapp# sudo python3 node.py
Applied Name prefix update successfully: (Advertise: /ndn/h5-site/h5/ofndn)

 ************* Node Status Changes *********

--------Sent <<<Helloreq>>> Interest --------
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--/ndn/h5-site/h5/ofndn/feature

--------Received <<<HelloRes>>> Data --------
 this is the <<<hello response>>> data

/ndn/h5-site/h5/ofndn
++++++++ Received <<<FeatureReq>>> interest ++++++++

++++++++ Sent <<<FeatureRes>>> Data ++++++++
```

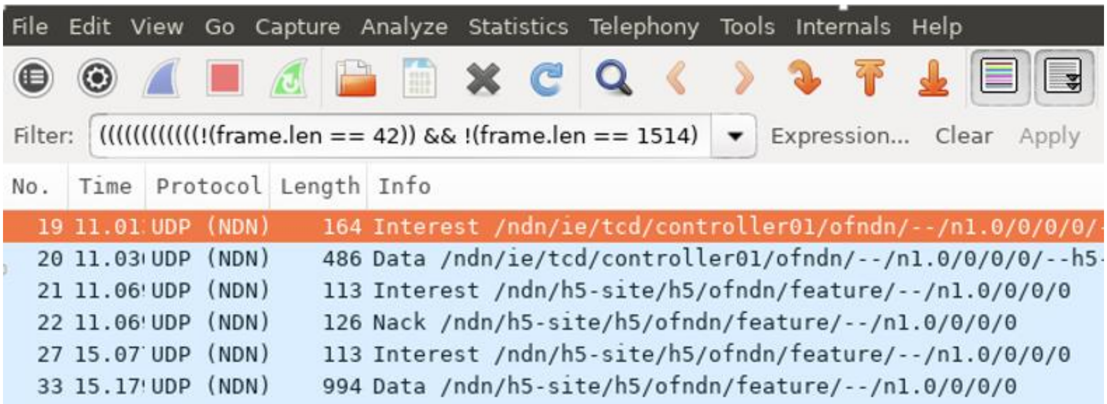
Figure 6.1: Switch Status Monitoring Function Running Result



```
"Node: h6"
root@ubuntu:~/PycharmProjects/ndnapp# sudo python3 controller.py
Applied Name prefix update successfully: (Advertise: /ndn/ie/tcd/controller01/ofndn)

 --------Received <<<HelloReq>>> Interest --------
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--/ndn/h5-site/h5/ofndn/feature
++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

TIMEOUT #1: /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

++++++++ Received <<<FeatureRes>>> Data ++++++++
========FIB Database has been updated=======
========Face Database has been updated=======

=================NPT=================
[['h5' '200002' '/n1.0/0/0/0/' '/ndn/h5-site/h5/ofndn/feature']]
-------------------------------------
```

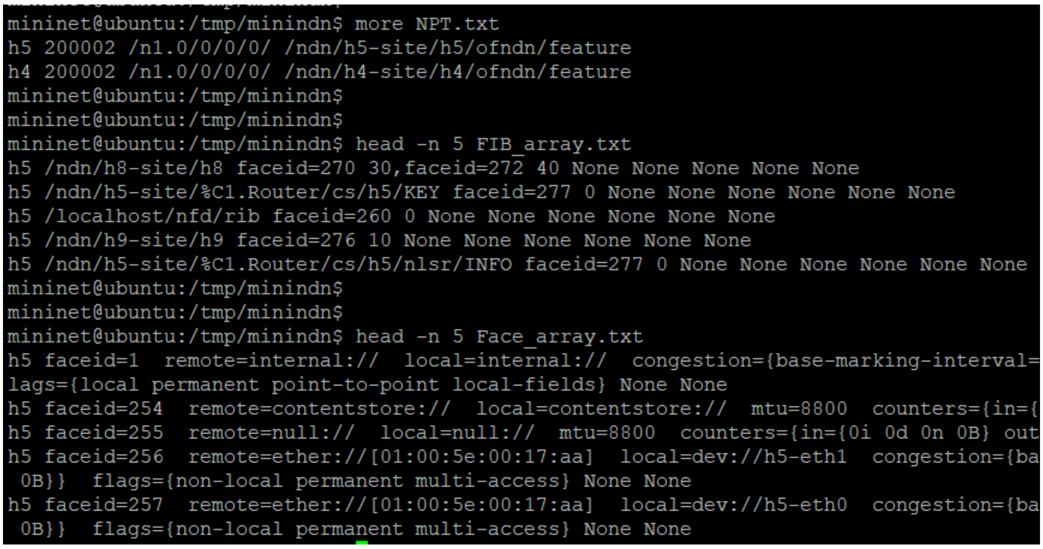
Figure 6.2: Controller Feature Updating Function Running Result

By analyzing the experimental results, we conclude that the model can achieve the desired function. It has the ability to monitor the status of the switch, and the controller can get the latest feature information of switches in time.



```
Filter: ((((((((((((!(frame.len == 42)) && !(frame.len == 1514)

No.  Time   Protocol  Length  Info
 19 11.01  UDP (NDN)     164 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/
 20 11.03  UDP (NDN)     486 Data /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5
 21 11.06  UDP (NDN)     113 Interest /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
 22 11.06  UDP (NDN)     126 Nack /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
 27 15.07  UDP (NDN)     113 Interest /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
 33 15.17  UDP (NDN)     994 Data /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
```

Figure 6.3: Packets for Status Monitoring and Feature Updating Functions



```
mininet@ubuntu:/tmp/minindn$ more NPT.txt
h5 200002 /n1.0/0/0/0/ /ndn/h5-site/h5/ofndn/feature
h4 200002 /n1.0/0/0/0/ /ndn/h4-site/h4/ofndn/feature
mininet@ubuntu:/tmp/minindn$
mininet@ubuntu:/tmp/minindn$
mininet@ubuntu:/tmp/minindn$ head -n 5 FIB_array.txt
h5 /ndn/h8-site/h8 faceid=270 30,faceid=272 40 None None None None None
h5 /ndn/h5-site/%C1.Router/cs/h5/KEY faceid=277 0 None None None None None None
h5 /localhost/nfd/rib faceid=260 0 None None None None None None
h5 /ndn/h9-site/h9 faceid=276 10 None None None None None None
h5 /ndn/h5-site/%C1.Router/cs/h5/nlsr/INFO faceid=277 0 None None None None None None
mininet@ubuntu:/tmp/minindn$
mininet@ubuntu:/tmp/minindn$
mininet@ubuntu:/tmp/minindn$ head -n 5 Face_array.txt
h5 faceid=1  remote=internal://  local=internal://  congestion={base-marking-interval=
lags={local permanent point-to-point local-fields} None None
h5 faceid=254  remote=contentstore://  local=contentstore://  mtu=8800  counters={in={
h5 faceid=255  remote=null://  local=null://  mtu=8800  counters={in={0i 0d 0n 0B} out
h5 faceid=256  remote=ether://[01:00:5e:00:17:aa]  local=dev://h5-eth1  congestion={ba
 0B}}  flags={non-local permanent multi-access} None None
h5 faceid=257  remote=ether://[01:00:5e:00:17:aa]  local=dev://h5-eth0  congestion={ba
 0B}}  flags={non-local permanent multi-access} None None
```

Figure 6.4: Output of NPT and FDB

- **Function Module 2: Configuration Request and Response Function in the switch and the controller.**

  Design Principle:

  The process of this function module is also related to both the switch and the controller. After the switch starts, it sends a CtrlInfoReq message, which has a long lifetime, to the controller. The controller holds it until new configuration information needs to be sent to this switch. Then it responds to the switch with a CtrlInfoRes message. After receiving the configuration information, the switch sends a new CtrlInfoReq message to the controller to request new configuration information.

  Evaluation Methods:

  To test this function module, we start a switch and a controller. After they have finished booting, we check the messages of CtrlInfoReq. Then we give the configuration information which needs to be sent to the switch. After that, we recheck the messages. Also, we capture and analyze the packets they use to communicate.

  Evaluation Result:

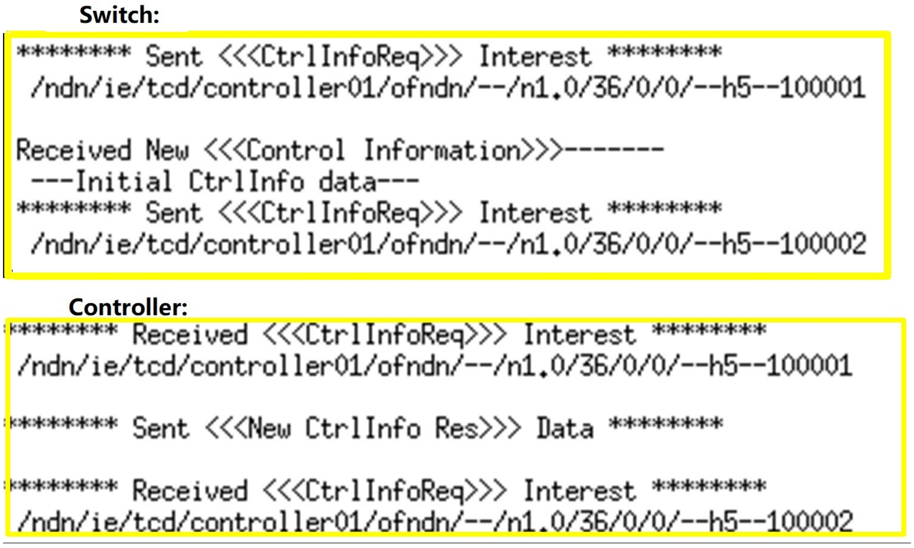
  Figure 6.5 displays the experimental results of the switch and controller. Figure 6.6 exhibits the relevant data packets captured via Wireshark.

  By analyzing the experimental results, we conclude that this function module can achieve the desired function. The switch can get configuration information from the controller at any time if necessary.

- **Function Module 3: Route Inquiring Function in the switch and Route Responding Function in the controller.**

  Design Principle:

  When the NDN node cannot forward an Interest packet, it sends a PacketIn message to the controller to query the route. After receiving the PacketIn, the con-



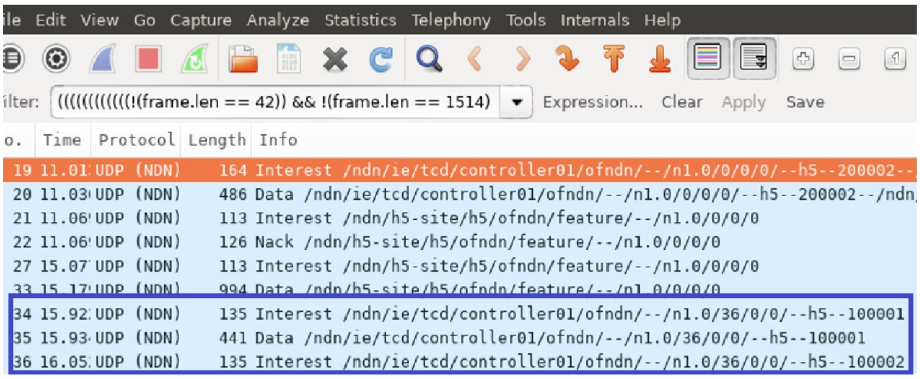
Switch:

```
******** Sent <<<CtrlInfoReq>>> Interest ********
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001

Received New <<<Control Information>>>-------
 ---Initial CtrlInfo data---
******** Sent <<<CtrlInfoReq>>> Interest ********
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100002
```

Controller:

```
******** Received <<<CtrlInfoReq>>> Interest ********
/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001

******** Sent <<<New CtrlInfo Res>>> Data ********

******** Received <<<CtrlInfoReq>>> Interest ********
/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100002
```

Figure 6.5: Configuration Request and Response Function Running Result


ile Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression...
Clear
Apply
Save

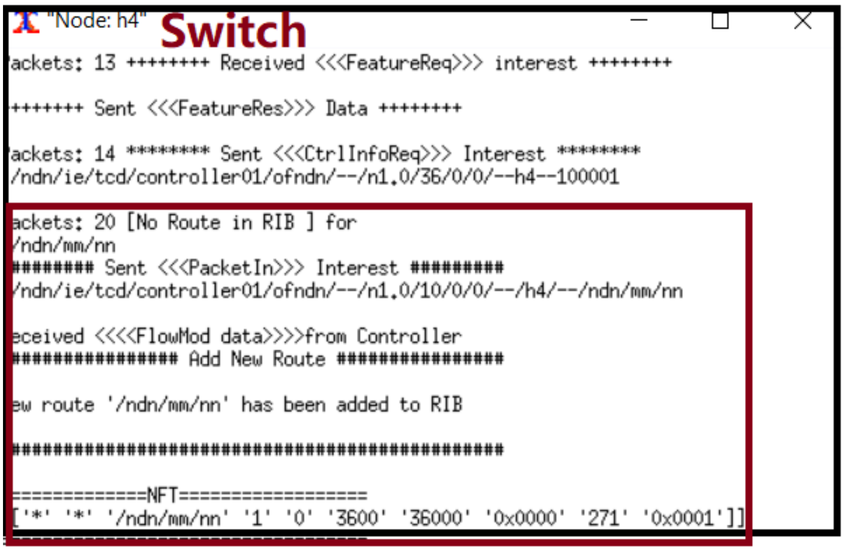
```
ilter: ((((((((((((!(frame.len == 42)) && !(frame.len == 1514)

o.  Time   Protocol Length Info
19 11.01 UDP (NDN)     164 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--
20 11.03 UDP (NDN)     486 Data /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--/ndn
21 11.06 UDP (NDN)     113 Interest /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
22 11.06 UDP (NDN)     126 Nack /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
27 15.07 UDP (NDN)     113 Interest /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
33 15.17 UDP (NDN)     994 Data /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
34 15.92 UDP (NDN)     135 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001
35 15.93 UDP (NDN)     441 Data /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001
36 16.05 UDP (NDN)     135 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100002
```

Figure 6.6: Packets for Configuration Request and Response Functions

troller sends a FlowMod message to instruct the switch to modify its FlowTable. In this way, the NDN node can forward the previous Interest packet then.

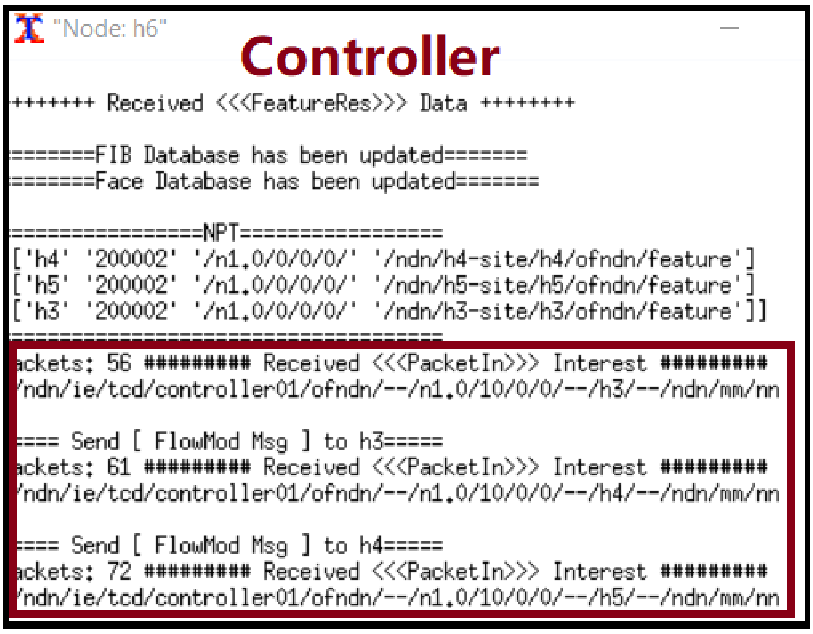
Evaluation Methods:
In this case, we use the implementation topology (Figure 5.1). Firstly, we start the two switches and the controller. Then we start a producer which advertises a name prefix, like “/ndn/mm/nn” and offers data for this prefix. The switches have no route to forward the Interest packet whose name is “/ndn/mm/nn”, so they have to ask the controller for help. After that, we run the consumer which sends an Interest packet with this name prefix. Finally, we check if the consumer gets the Data packet from the producer, and analyze the packets captured via Wireshark.

Evaluation Result:
Figure 6.7 and 6.8 exhibit the results of this function running on the switch and the controller. Figure 6.9 gives the packets exchanged between the controller and one switch. And the experiment result shows that the consumer gets the Data packet finally. So, it proves that this module can achieve its goal, and our module has the functions of route inquiring and route adjusting.



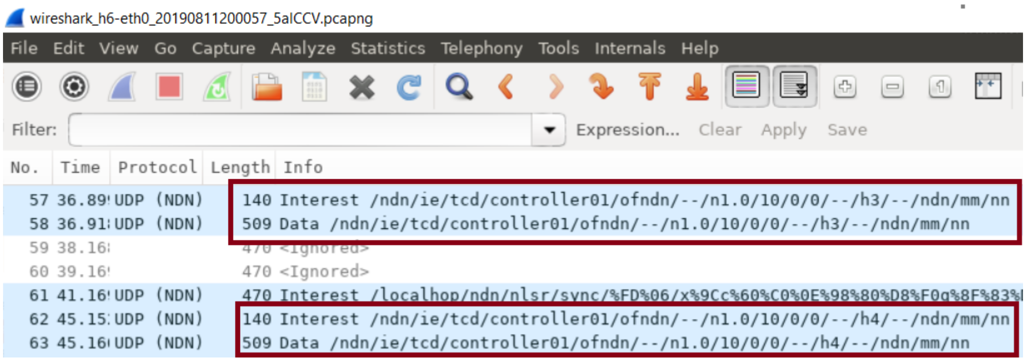
```
ackets: 13 ++++++++ Received <<<FeatureReq>>> interest ++++++++

+++++++ Sent <<<FeatureRes>>> Data ++++++++

ackets: 14 ******** Sent <<<CtrlInfoReq>>> Interest ********
/ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h4--100001

ackets: 20 [No Route in RIB ] for
/ndn/mm/nn
######## Sent <<<PacketIn>>> Interest #########
/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h4/--/ndn/mm/nn

eceived <<<<FlowMod data>>>>from Controller
################ Add New Route ################

ew route '/ndn/mm/nn' has been added to RIB

###############################################

============NFT=================
['*' '*' '/ndn/mm/nn' '1' '0' '3600' '36000' '0x0000' '271' '0x0001']]
```

Figure 6.7: Switch Route Inquiring Function Running Result



```
+++++++ Received <<<FeatureRes>>> Data ++++++++

=======FIB Database has been updated=======
=======Face Database has been updated=======

================NPT================
[['h4' '200002' '/n1.0/0/0/0/' '/ndn/h4-site/h4/ofndn/feature']
 ['h5' '200002' '/n1.0/0/0/0/' '/ndn/h5-site/h5/ofndn/feature']
 ['h3' '200002' '/n1.0/0/0/0/' '/ndn/h3-site/h3/ofndn/feature']]
====================================
ackets: 56 ######### Received <<<PacketIn>>> Interest #########
/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h3/--/ndn/mm/nn

==== Send [ FlowMod Msg ] to h3=====
ackets: 61 ######### Received <<<PacketIn>>> Interest #########
/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h4/--/ndn/mm/nn

==== Send [ FlowMod Msg ] to h4=====
ackets: 72 ######### Received <<<PacketIn>>> Interest #########
/ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h5/--/ndn/mm/nn
```

Figure 6.8: Controller Route Responding Function Running Result



wireshark_h6-eth0_20190811200057_5alCCV.pcapng
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save

```
No.  Time   Protocol  Length Info
 57 36.89 UDP (NDN)   140 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h3/--/ndn/mm/nn
 58 36.91 UDP (NDN)   509 Data /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h3/--/ndn/mm/nn
 59 38.16             470 <Ignored>
 60 39.16             470 <Ignored>
 61 41.16 UDP (NDN)   470 Interest /localhop/ndn/nlsr/sync/%FD%06/x%9Cc%60%C0%0E%98%80%D8%F0g%8F%83%
 62 45.15 UDP (NDN)   140 Interest /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h4/--/ndn/mm/nn
 63 45.16 UDP (NDN)   509 Data /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h4/--/ndn/mm/nn
```

Figure 6.9: Packets for Route Inquiring and responding Functions

- **Function Module 4: Switch's other functions. Design Principle:**

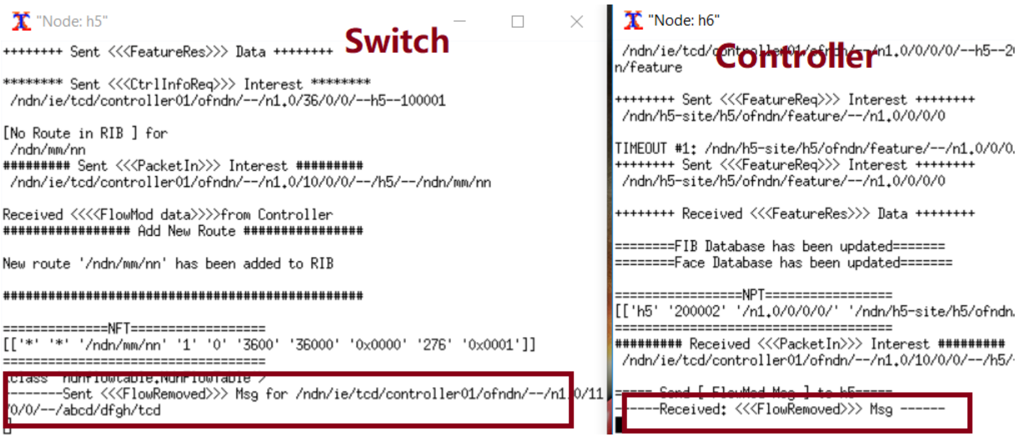
  Other functional modules of the switch can send messages for specific purposes to the controller, for example, FlowRemoved and Error. At present, these functions are relatively simple, and the packet can be sent directly to the name prefix the controller designated.

  Evaluation Methods:

  Simulate sending this type of message to the controller from a switch and check the results.

  Evaluation Result:

  Figure6.10 shows that this function works correctly. The switch has the ability to report Error and FlowRemoved to the controller.



```
++++++++ Sent <<<FeatureRes>>> Data ++++++++

******** Sent <<<CtrlInfoReq>>> Interest ********
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001

[No Route in RIB ] for
 /ndn/mm/nn
######### Sent <<<PacketIn>>> Interest #########
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h5/--/ndn/mm/nn

Received <<<<FlowMod data>>>>from Controller
################# Add New Route ################

New route '/ndn/mm/nn' has been added to RIB

################################################

=============NFT=================
[['*' '*' '/ndn/mm/nn' '1' '0' '3600' '36000' '0x0000' '276' '0x0001']]
=================================
<class 'ndnflowtable.NdnFlowTable'>
--------Sent <<<FlowRemoved>>> Msg for /ndn/ie/tcd/controller01/ofndn/--/n1.0/11
/0/0/--/abcd/dfgh/tcd
```

```
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--2
n/feature

++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

TIMEOUT #1: /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0.
++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

++++++++ Received <<<FeatureRes>>> Data ++++++++

========FIB Database has been updated=======
========Face Database has been updated=======

=================NPT=================
[['h5' '200002' '/n1.0/0/0/0/' '/ndn/h5-site/h5/ofndn.
=====================================
######### Received <<<PacketIn>>> Interest #########
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/10/0/0/--/h5/-

=====Send [ FlowMod Msg ] to h5=====
-------Received: <<<FlowRemoved>>> Msg ------
```

Figure 6.10: FlowRemoved Message Delivery

- **Function Module 5: Controller's other functions.**

  Design Principle:

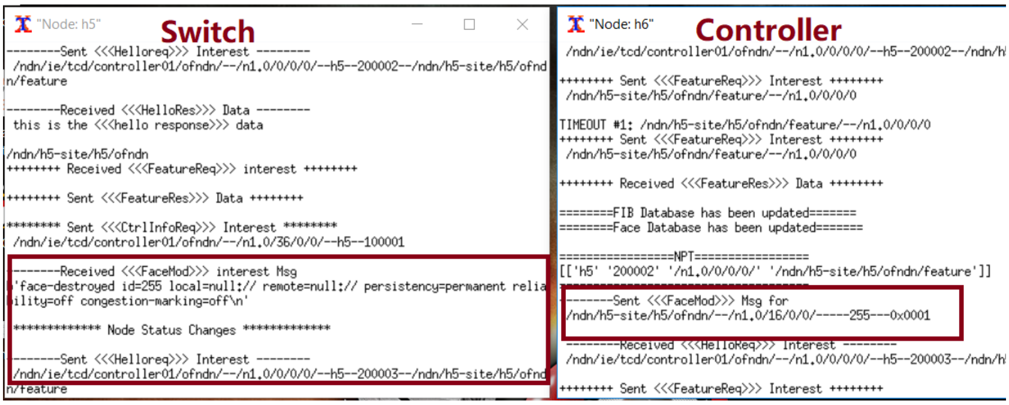
  Other functional modules of the controller can send messages for specific purposes to the switches, for example, FaceMod and PacketOut. At present, these functions are relatively simple, and these packets can be sent directly to the name prefix the switch indicated.

Evaluation Methods:

Simulate sending this type of message to a switch from the controller and check the results.

Evaluation Result:

Figure 6.11 shows that this function works correctly. The controller has the ability to send FaceMod and PacketOut messages to switches.



```
--------Sent <<<Helloreq>>> Interest --------
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--/ndn/h5-site/h5/ofnd
n/feature

--------Received <<<HelloRes>>> Data --------
 this is the <<<hello response>>> data

/ndn/h5-site/h5/ofndn
++++++++ Received <<<FeatureReq>>> interest ++++++++

++++++++ Sent <<<FeatureRes>>> Data ++++++++

******** Sent <<<CtrlInfoReq>>> Interest ********
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/36/0/0/--h5--100001

-------Received <<<FaceMod>>> interest Msg
'face-destroyed id=255 local=null:// remote=null:// persistency=permanent relia
ility=off congestion-marking=off\n'

************* Node Status Changes *************

-------Sent <<<Helloreq>>> Interest --------
/ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200003--/ndn/h5-site/h5/ofnd
n/feature
```

```
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200002--/ndn/h

++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

TIMEOUT #1: /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0
++++++++ Sent <<<FeatureReq>>> Interest ++++++++
 /ndn/h5-site/h5/ofndn/feature/--/n1.0/0/0/0

++++++++ Received <<<FeatureRes>>> Data ++++++++

========FIB Database has been updated=======
========Face Database has been updated=======

=================NPT=================
[['h5' '200002' '/n1.0/0/0/0/' '/ndn/h5-site/h5/ofndn/feature']]

-------Sent <<<FaceMod>>> Msg for
/ndn/h5-site/h5/ofndn/--/n1.0/16/0/0/-----255---0x0001

 --------Received <<<HelloReq>>> Interest --------
 /ndn/ie/tcd/controller01/ofndn/--/n1.0/0/0/0/--h5--200003--/ndn/h

++++++++ Sent <<<FeatureReq>>> Interest ++++++++
```

Figure 6.11: FaceMod Message Delivery

6.3.2 Scalability Evaluation

The previous section used the basic implementation topology and made a comprehensive assessment of the functionality of our model. This section aims to evaluate its scalability, so a large and more complex topology is needed. We run our model on a representative network environment to verify its scalability.

Design Principle:

Our model works on the NDN, which relies on the underlying communication of the NDN. In addition, its performance is also related to the hardware platform it is hosted. In theory, as long as the hardware and network are not limited, our model can be deployed to any complex network environment. On the switch side, only the switch agent needs to be run to manage the local NDN node and to communicate with the controller, all of which consume fewer resources. On the controller side, with the growth of the number of the managed switches, the required resources increase continuously, such as memory resources, computing resources, and network bandwidth resources. Therefore, the extensibility limitation of this model is the platform on which it runs, not its own architectural design.

Evaluation Methods:

In this case, we use Figure 6.12 as the experimental topology. We add three switches to the original Implementation Topology and make the network structure more complex. This topology contains multi-path, loops, and the indirectly connected network devices mentioned in the previous section. Therefore, this topology represents the network deployment architecture commonly used in real-world environments. Furthermore, we boot up the controller and some of the switches first, after they are running stably and the network are converged, we start the remaining switches to test the stability of the model.

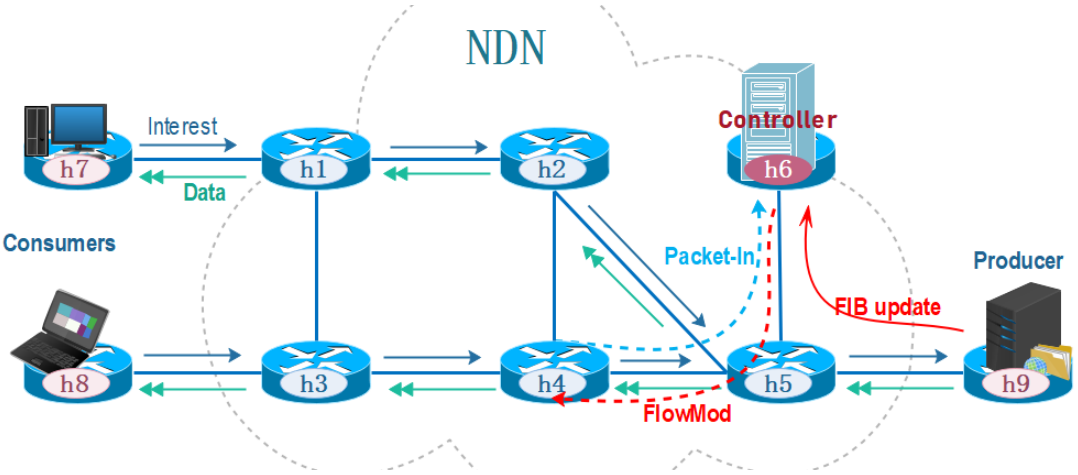
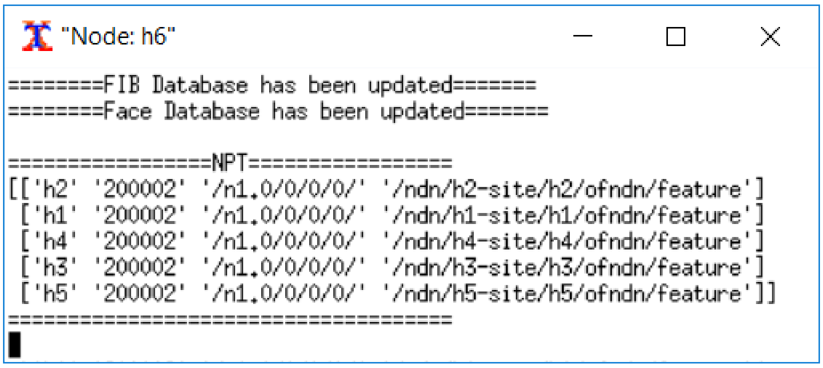


Figure 6.12: Large Scale Topology

Evaluation Result:

Experimentally, the model can still work correctly in a complex network environment. Figure 6.13 shows the managed switches by the controller. In addition, during the running of the model, the increase and decrease of the managed equipment will not affect the stability of other devices and the entire system. Moreover, we can add more

NDN nodes for testing, but it is not necessary, because the topology we use is complex enough to represent the typical situation in the real environment. adding more devices does not make much sense, but it only consumes more resources of the controller. In summary, we confirm that the model has excellent scalability.



```
========FIB Database has been updated=======
========Face Database has been updated=======

=================NPT=================
[['h2' '200002' '/n1.0/0/0/0/' '/ndn/h2-site/h2/ofndn/feature']
 ['h1' '200002' '/n1.0/0/0/0/' '/ndn/h1-site/h1/ofndn/feature']
 ['h4' '200002' '/n1.0/0/0/0/' '/ndn/h4-site/h4/ofndn/feature']
 ['h3' '200002' '/n1.0/0/0/0/' '/ndn/h3-site/h3/ofndn/feature']
 ['h5' '200002' '/n1.0/0/0/0/' '/ndn/h5-site/h5/ofndn/feature']]
=====================================
```

Figure 6.13: Node Prefix Table for large scale topolgy

6.3.3 Compatibility Evaluation

In the switch architecture design in section 4.1.1, we also considered compatibility. We reserved a field to store the traditional IP flow information when we designed the FlowTable, so this FlowTable can be used for IP flow forwarding. For example, a switch running on the NDN needs to forward NDN packets and also forward IP packets at the same time. Then its FlowTable can include both the FlowEntries for NDN packet and the FlowEntries for IP flow, but the actions of these FlowEntries are different. The action of IP FlowEntry can be ’forward to a specific interface’, and the action of NDN FlowEntry might be ’forward to a specific face’. Its ”heterogeneous” FlowTable is managed by the controller via NOF protocol, and the routing decision is passed to upper applications.

The message type of the NOF protocol we have designed so far is mainly for NDN packet forwarding. In the future, we can add more NOF message types to better support IP packet forwarding. For example, we can add an InterfaceMod message to modify the state of an interface.

In brief, our model is mainly designed for NDN packet forwarding, but it is also compatible with the forwarding of traditional IP packets. Further, more functions can be added according to actual needs to better compatible with traditional IP flow forwarding. So our model has good compatibility.

6.3.4 Security and Privacy Evaluation

Security and privacy are not independent, simple, and negligible concepts. They involve a series of interdependent, interacting monolithic properties of a protocol or system. With the high popularity of digital technology nowadays, people have long recognized the importance of security and privacy. All RFC documents are required to consider security and privacy by RFC 3552[59] and RFC 6973[60], the other documents relate to digital technology are also encouraged to contain the security section. In our architecture, we combine two fundamental technologies together. So, both of them should be considered in security and privacy aspects. Since security and privacy are relative concepts, we do not intend to describe them separately in this section.

We first consider the security of the underlying protocol of this model. At the beginning of the CCN and NDN design, security was of particular concern. [10] and [9] provide a detailed introduction of CCN and NDN network security. In short, security in NDN is data-centric, which means all security issues are related to data content. In NDN, the confidentiality of data is protected by encryption, and all data is authenticated with digital signatures, which can ensure data's reliability and integrity. Unlike the IP security, it is not necessary to verify the data source in NDN. As long as the reliability and integrity of the data are guaranteed, the data source is not important. In addition, the NDN data transmission mechanism makes it fundamentally avoid many security threats, such as man-in-the-middle attacks, denial-of-service, DNS spoofing. Our model runs over the NDN and transmits packets through the NDN, so the underlying transmission of this model is secure.

Then we consider the security of the model itself. The components involved in our model are only switches and controller. Since the packets can be distinguished via digital signature, then it is possible to use digital signatures to implement access control. For example, the controller can set different access policies for different signed packets or filter out undesired access. Besides, since our model uses NDN packets, it can also benefit from NDN security.

Overall, the security of our model benefits from the security mechanisms of the NDN, and it can also perform access control based on the characteristics of NDN packets. So our model has high-grade security

6.4 Summary

This chapter provides a comprehensive assessment of the NOF-SDN model. The first section summarizes the contents that need to be evaluated, including the functional modules, scalability, compatibility, and security. The evaluation approaches are introduced in the next section. Section 3 gives the details of the evaluation process and the results. It first focuses on evaluating the various functional modules, including the functions of the switch and controller. Then it assesses the scalability of the model by using a more extensive, more complex topology. Finally, compatibility and security are also evaluated and analyzed.

In conclusion, all the functions of this model can work correctly, and it is quite capable of network management and route control. It also has good scalability, compatibility, and security.

Chapter 7

Conclusion

“NOF-SDN” is an OpenFlow based SDN implementation over NDN. This study refers to the traditional OpenFlow protocol and designs the NOF-SDN architecture, which aims to apply the centralized network management and routing decision functions of SDN to the emerging NDN. This chapter summarizes the achievement and contributions, then concludes with a discussion of its limitations and future work.

7.1 Achievement

This study analyzes two emerging network architectures, SDN and NDN, and discovers the feasibility and advantages of combining these two architectures.SDN provides centralized management and routing decisions for traditional networks. NDN uses content-centric architecture to improve transmission efficiency and security. Although these two architectures are different ways of solving the problems of the conventional IP network architecture, they are not competitive. They can be fused and leverage each other’s strengths. The industry has recognized this, and many integration solutions have been proposed. This study examined the existing fusion schemes in detail and found their shortcomings. For short-term solutions, they mainly use traditional SDN architecture to manage NDN nodes. This solution relies on traditional IP networks and does not benefit from the efficient forwarding of NDN networks. For long-term solutions, they completely abandoned the existing SDN architecture and redesigned the new model for NDN network management, which is very complicated and expensive, and does not have compatibility with traditional IP networks.

For the above reasons, this study designed a new fusion solution, NOF-SDN, and developed a prototype. The NOF-SDN architecture design includes the structural design, functional design, and communication message design of the switch and controller. The prototype can implement status monitoring, configuration acquisition, routing query, and other functions. Moreover, the model runs directly on the NDN without relying on traditional IP networks, and it is compatible with managing traditional IP routes. Its scalability and security are also reflected in the design. This prototype was then deployed to the simulated NDN environment, and its functions were evaluated in detail to demonstrate its feasibility, functionality, compatibility, and security. The final source code is shared on GitHub. Compared with the existing SDN and NDN convergence schemes, this design refers to the traditional OpenFlow protocol as much as possible. It can run directly on the NDN network without IP network support and is also compatible with the management of traditional IP flows.

7.2 Future Works

This section discusses the limitation of this research project and proposes the directions of future works.

- **Limited network management capability**
  Due to the limited time of the project, this model only implements basic management of the NDN node, such as modifying its FlowTable, state of a face, or FIB. However, these management functions are relatively simple. And the monitoring capability of the node only stays in the aspect of monitoring its state change. These features are not enough for centralized management and unified deployment of the network. In the future, we can add more features to achieve better network management. For example, large-scale, unified deployment of switch networks through controllers and more comprehensive network monitoring, including monitoring of node software and hardware resources and usage. Of course, this is a complex system, and the function of NDF's open NFD API is limited. More comprehensive network management is bound to modify the design and function of the underlying NFD. So this is a long-term plan.

- **Integrate upper applications**
  Although this model is only responsible for the protocol between the switch and the controller, it needs to integrate with the upper applications, so that the network management and routing decision functions can work properly. The design of upper applications is out of the scope of this research topic, but it requires a better, unified interface for combining the upper applications. In future work, we can design a more standardized API for integrating upper applications.

- **Improve Security**
  In Section 6.3.4, we theoretically evaluated the security of this model, but we have not actually applied it to this model. PyNDN2 provides some security methods by default, such as the Interest packet digest signature. Some other security-related features are still under development. In future work, more security policies and mechanisms can be used in this model. For example, design a trust management mechanism, an encryption mechanism.

Bibliography

[1] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, "Software-defined networking (sdn): a survey," *Security and communication networks*, vol. 9, no. 18, pp. 5803–5833, 2016.

[2] "Network function virtualization." `https://www.etsi.org/technologies/nfv`, Online, accessed 18/07/2019.

[3] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012.

[4] R. Dantu, T. A. Anderson, R. Gopal, and L. L. Yang, "Forwarding and control element separation (forces) framework," *RFC: 3746*, 2004.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[6] "Open networking foundation press release." `https://web.archive.org/web/20110326024026/http://www.opennetworkingfoundation.org/?p=7`, Online, accessed 18/07/2019.

[7] "Openflow messages in flowgrammable.org." `http://flowgrammable.org/sdn/openflow/message-layer/`, Online, accessed 25/07/2019.

[8] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[9] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang, *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[10] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pp. 1–12, ACM, 2009.

[11] "Information-centric networking research group." `https://irtf.org/icnrg`, Online, accessed 18/07/2019.

[12] "Ndn frequently asked questions (faq)." `https://named-data.net/project/faq/`, Online, accessed 20/07/2019.

[13] "National science foundation(nsf)." `http://www.nets-fia.net/`, Online, accessed 29/07/2019.

[14] "Ndn project overview." `https://named-data.net/project/`, Online, accessed 20/07/2019.

[15] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[16] "Sdn in flowgrammable.org." `http://flowgrammable.org/sdn/`, Online, accessed 20/07/2019.

[17] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks: an authoritative review of network programmability technologies.* " O'Reilly Media, Inc.", 2013.

[18] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible openflow-controller benchmark," in *2012 European Workshop on Software Defined Networking*, pp. 48–53, IEEE, 2012.

[19] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[20] "Opendaylight." `https://www.opendaylight.org/`, Online, accessed 23/07/2019.

[21] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control plane of software defined networks: A survey," *Computer communications*, vol. 67, pp. 1–10, 2015.

[22] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 13–18, ACM, 2013.

[23] "The open networking foundation (onf)." `https://www.opennetworking.org/tag/openflow/`, Online, accessed 23/07/2019.

[24] "Openflow protocol in flowgrammable.org." `http://flowgrammable.org/sdn/openflow/#tab_protocol`, Online, accessed 25/07/2019.

[25] M. Dusi, R. Bifulco, F. Gringoli, and F. Schneider, "Reactive logic in software-defined networking: Measuring flow-table requirements," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 340–345, IEEE, 2014.

[26] "Ndn architecture." `https://named-data.net/project/archoverview/`, Online, accessed 26/07/2019.

[27] "Ndn protocol design principles." `https://named-data.net/project/ndn-design-principles/`, Online, accessed 26/07/2019.

[28] "Ndn packets." `https://named-data.net/doc/NDN-packet-spec/current/`, Online, accessed 26/07/2019.

[29] "Ndn specifications contenttype." `https://redmine.named-data.net/projects/ndn-tlv/wiki/ContentType`, Online, accessed 26/07/2019.

[30] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A case for stateful forwarding plane," *Computer Communications*, vol. 36, no. 7, pp. 779–791, 2013.

[31] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, *et al.*, "Named data networking (ndn) project," *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, vol. 157, p. 158, 2010.

[32] "Ndn platform." `https://named-data.net/codebase/platform/`, Online, accessed 29/07/2019.

[33] "Nfd documentation." `http://named-data.net/doc/NFD/0.6.5/`, Online, accessed 29/07/2019.

[34] "Nfd developer resources." `https://redmine.named-data.net/projects/nfd/wiki#NFD-Developer-Resources`, Online, accessed 29/07/2019.

[35] A. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "Nlsr: named-data link state routing protocol," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pp. 15–20, ACM, 2013.

[36] "Nlsr documentation." `http://named-data.net/doc/NLSR/current/`, Online, accessed 29/07/2019.

[37] V. Lehman, A. M. Hoque, Y. Yu, L. Wang, B. Zhang, and L. Zhang, "A secure link state routing protocol for ndn," *Tech. Rep. NDN-0037*, 2016.

[38] "Nlsr githup repository." `https://github.com/named-data/NLSR`, Online, accessed 29/07/2019.

[39] Q.-Y. Zhang, X.-W. Wang, M. Huang, K.-Q. Li, and S. K. Das, "Software defined networking meets information centric networking: A survey," *IEEE Access*, vol. 6, pp. 39547–39563, 2018.

[40] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri, "Information centric networking over sdn and openflow: Architectural aspects and experiments on the ofelia testbed," *Computer Networks*, vol. 57, no. 16, pp. 3207–3221, 2013.

[41] G. Siracusano, S. Salsano, P. L. Ventre, A. Detti, O. Rashed, and N. Blefari-Melazzi, "A framework for experimenting icn over sdn solutions using physical and virtual testbeds," *Computer Networks*, vol. 134, pp. 245–259, 2018.

[42] A. Detti, N. Blefari Melazzi, S. Salsano, and M. Pomposini, "Conet: a content centric inter-networking architecture," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pp. 50–55, ACM, 2011.

[43] A. Detti, M. Pomposini, N. Blefari-Melazzi, and S. Salsano, "Supporting the web with an information centric network that routes by name," *Computer Networks*, vol. 56, no. 17, pp. 3705–3722, 2012.

[44] S. Signorello, R. State, J. François, and O. Festor, "Ndn. p4: Programming information-centric data-planes," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pp. 384–389, IEEE, 2016.

[45] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[46] I. Moiseenko and D. Oran, "Tcp/icn: Carrying tcp over content centric and named data networks," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, pp. 112–121, ACM, 2016.

[47] S. Luo, S. Zhong, and K. Lei, "Ip/ndn: A multi-level translation and migration mechanism," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5, IEEE, 2018.

[48] G. Xylomenos, Y. Thomas, X. Vasilakos, M. Georgiades, A. Phinikarides, I. Doumanis, S. Porter, D. Trossen, S. Robitzsch, M. J. Reed, *et al.*, "Ip over icn goes live," in *2018 European Conference on Networks and Communications (EuCNC)*, pp. 319–323, IEEE, 2018.

[49] B. E. Rajendran Jeeva, "Openflow-based control plane for information-centric networking," in *OpenFlow-based control plane for Information-Centric Networking)*, TCD Master Dissertation, 2016.

[50] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, "Information-centric networking: seeing the forest for the trees," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, p. 1, ACM, 2011.

[51] O. TS-015, "Openflow switch specification." https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.3.pdf, Online, accessed 05/08/2019.

[52] "Openflow error message error type." http://flowgrammable.org/sdn/openflow/message-layer/error/#Error_1.4, Online, accessed 05/08/2019.

[53] "Nfd management protocol." https://redmine.named-data.net/projects/nfd/wiki/Management, Online, accessed 06/08/2019.

[54] "Minindn simulator." http://minindn.memphis.edu/index.php, Online, accessed 08/08/2019.

[55] "Ndn platform." http://named-data.net/codebase/platform/, Online, accessed 08/08/2019.

[56] "Minindn installation." http://minindn.memphis.edu/mini_ndn_installation.phpp, Online, accessed 08/08/2019.

[57] "Pyndn2 documentation." https://github.com/named-data/PyNDN2/blob/master/README.md, Online, accessed 08/08/2019.

[58] "Wireshark ndn plugin." https://github.com/named-data/ndn-tools/tree/master/tools/dissect-wireshark, Online, accessed 08/08/2019.

[59] E. Rescorla and B. Korver, "Rfc 3552: Guidelines for writing rfc text on security considerations," *Internet Society Req. for Comm*, 2003.

[60] A. Cooper, H. Tschofenig, B. Aboba, J. Peterson, J. Morris, M. Hansen, and R. Smith, "Rfc 6973: Privacy considerations for internet protocols," *IETF. Retrieved from tools. ietf. org/html/rfc6973*, 2013.