OF-ICN:

OpenFlow-based control plane for Information-Centric Networking

by

Rajendran Jeeva, B E

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2016

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.

Rajendran Jeeva

August 31, 2016

Permission to Lend and/or Copy

Dissertation Summary

Software-Defined Networking(SDN) is a networking architecture, that is proposed to decouple the complex control functions out of the network switching elements and form a separate control layer. Thus, the infrastructure in network elements can carry out less complex forwarding operations and the control layer can programme this infrastructure, based on the control policies directed by the network applications. *OpenFlow* is an SDN approach, that provides a centralised controller for the underlying OpenFlow switches. OpenFlow switches are equipped with flow tables, that can be managed by the controller through an interface,*OpenFlow protocol.* OpenFlow yields a lot of benefits to the network, including, a global view of the network for better network control, flow-based traffic control, clear separation between the infrastructure and the control layer for easy conversion of business needs into low-level instructions and proactive management decisions. *Information-Centric Networking*(ICN) is a future internet architecture aims to shift the internet architecture from *'host-centric'* to *'content-centric'*, by keeping the content as the first primitive in network communication, instead of addressing the end-points. ICN is gaining a lot of interest from the industry and academic research due to its ability to solve a number of performance issues, that are faced by today's Internet due to the enormous increase in the amount of data being handled on the Internet. ICN provides a number of benefits, including, mobility, secured contents, improved content availability and multicast communication. Recently, researchers commenced the idea of integrating OpenFlow and ICN in order to practice the benefits of one over the other. There are some thoughts in literature to integrate OpenFlow and ICN, without modifying any of the architectures under consideration. They involve workarounds and plugins, that need a lot of efforts and fail to provide a clean state integration of OpenFlow and ICN. On the other hand, a number of studies tried to extend OpenFlow to some extent, to enable ICN features, but they lack in the amount of ICN functionalities that are supported and the scope of the extensions was minimal. Most of the studies remain conceptual. This dissertation aimed to study the potential of OpenFlow to support ICN, by creating an OpenFlow-based control plane for the ICN network. This is achieved by designing an integration solution *'OF-ICN'*, by extending the necessary OpenFlow elements: OpenFlow switch, OpenFlow controller and OpenFlow protocol, with the ICN functionalities and ICN data structures. OF-ICN is implemented using an open source Python OpenFlow controller, POX. The OpenFlow switch component provided by POX is extended to include ICN-related data structures: *Forwarding Information Base, Pending Interest Table and Content Store*. Similarly, the OpenFlow controller component is extended with the name-based routing database and a cache to store contents. POX's OpenFlow library is modified to introduce new messages, actions and events, to be used between the switch and the controller. The implementation of OF-ICN is evaluated by benchmarking the ICN functionalities supported by it and also by comparing its behaviour with a number of existing ICN implementations. The results show that, with the necessary modifications to the underlying elements and library, a clean state integration between OpenFlow and ICN can be achieved and, OpenFlow can provide a control plane to manage the underlying ICN-enabled network. In addition to providing an experimental solution for integrating OpenFlow and ICN, this dissertation has also contributed towards a complete analysis of existing literature towards combining OpenFlow and ICN and reported the gaps identified in them. It also ensured to utilise the generic ICN architecture which can be adapted to any specific ICN implementation. A modularized code is produced out of this dissertation, which can be separated into modules and plugged into a relevant codebase. This study has identified the ICN naming scheme, scalability and security as future research directions over the proposed approach.

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. Stefan Weber, for his excellent guidance and motivation throughout the dissertation. I would also like to extend my thanks to my mum, who always holds me in my ups and downs. Many thanks to Nina, for her patience and continuous support.

Rajendran Jeeva

University of Dublin, Trinity College

September 2016

OF-ICN:

OpenFlow-based control plane for Information-Centric Networking

Rajendran Jeeva

MSc in Computer Science (Networks and Distributed Systems)

University of Dublin, Trinity College, 2016

Supervisor: Weber Stefan

OpenFlow is a Software-Defined Networking (SDN) approach, that is used to separate the data plane and the control plane of a network. This is achieved by defining and separating the network communication into different flows and by controlling the paths of these flows using OpenFlow. The work on the OpenFlow protocol currently focuses and relies on IP-based networking. Information-Centric Networking (ICN) is an alternative Internet architecture, that provides network communications based on the named contents instead of addresses as in current Internet architecture. However, current ICN approaches lack the definition of a control plane and current OpenFlow specifications do not support the control of ICN flow by default. Therefore, this study analyses the potential of OpenFlow in supporting ICN and tries to port ICN functionalities in OpenFlow by modifying and extending OpenFlow components; switch, controller and the protocol. The results from this study reveal that, upon making necessary modifications to the underlying protocol, OpenFlow can successfully support ICN functionalities by making forwarding and caching decisions for ICN flows in the network.

Contents

List of Tables

List of Figures

Dissertation Map

This dissertation is divided into following chapters:

- ***Chapter 1* -** This chapter introduces the reader to the motivation for this study. It also lists some background concepts and terminologies that are used in the remaining chapters of the dissertation

- ***Chapter 2* -** This chapter presents the past research and current on-going research activities in OpenFlow and ICN, giving particular attention to their integration. The research works are categorized into major sections that are addressed by this dissertation

- ***Chapter 3* -** This chapter compares the architectural differences between OpenFlow and ICN and formulates a solution, 'OF-ICN', for the gap in OpenFlow extension, which the previous studies failed to address

- ***Chapter 4* -** This chapter puts forward the architectural design of ICN enabled OpenFlow switch and control plane, by quoting the suggested extensions for the OpenFlow, in terms of new messages and actions

- ***Chapter 5* -** This chapter shows the implementation details of OF-ICN and briefs the technologies used for the implementation

- ***Chapter 6* -** This chapter evaluates the approach of OF-ICN, using the implemented experiment and comparing it with existing ICN prototypes. It also ensures that OF-ICN provides essential ICN functionalities

- ***Chapter 7* -** This chapter derives the conclusion based on the evaluation results shown in the previous chapter and also sets the future research directions

Chapter 1

Introduction

Software-Defined Networking(SDN) is a network architecture paradigm, that decouples the complex network control and management operations of the network elements from their forwarding operations[2]. These separated control and management functions will form a *control layer* for the underlying network elements. *OpenFlow* is an SDN approach, that follows the principles of SDN and, provides a centralised controller and a programming interface for the network switches, that are OpenFlow-enabled[3]. OpenFlow provides a clear separation between the control policies and the switch infrastructure so that the switches can be less complex and carry out the essential forwarding operations. OpenFlow divides the network communication into various *'flows'* to provide better control and traffic engineering [3]. *Information-Centric Networking*(ICN) is a future Internet architecture that shifts the network architecture from being *'host-centric'* to *'content-centric'* [4]. ICN provides network communication by keeping the *'data'* or *'content'* as the first primitive and enables the network to fetch the content from anywhere in the network using a *'name'* to identify the content [5]. ICN aims to provide better availability of content in the network resulting in better user experience and minimal network traffic to the content producer [4]. OpenFlow, by default, has the complete support for the IP-based network communication flows. However, none of the OpenFlow specifications has the support for controlling ICN flows [6, 7, 8, 9]. On the other side, ICN architectures have no definition of a centralised control plane to enhance the controlling process for the ICN networks [5]. There are some studies in literature, which tried integrating OpenFlow and ICN by either providing an overlay solution or providing workarounds on OpenFlow with extra modules or plugins for handling ICN flows [10, 11]. There are very fewer considerations to move OpenFlow towards a clean state ICN support [12, 13, 14], but the scope and the supported ICN functionalities are minimal. This dissertation studies the potential for the OpenFlow to support ICN by extending the essential OpenFlow elements with ICN features. To achieve this, this study aims to create an OpenFlow-based control plane using an OpenFlow controller, POX [15] and, modifies it based on the proposed solutions.

The following section outlines the background of OpenFlow and ICN, followed by the aims of this project. Next section presents an abstracted view of the literature outlining the gaps, followed by the approach proposed by this dissertation to fill those gaps. Specific project contributions are listed in the following section and this chapter concludes with a list of terminology that will be used throughout this report.

1.1 Background

This section describes the background details on SDN, OpenFlow and ICN.

1.1.1 SDN and OpenFlow for 'Today'

Network switches are the network elements that help to connect machines to the network and are essential in network operations between these machines [16]. The network operations include packet lookup, packet switching and packet buffering activities [1, 17]. A switch has two main components to carry out these operations: a *'datapath'* which is used to forward the packets in the network and a *'control block'* to make decisions on switch configurations and operations, which includes management protocols like SNMP [18] and, routing protocols like OSPF [19] and BGP [20]. These components are tightly coupled to a network switch, leading to a closed architecture, wherein the data plane and the control plane are packed together by the vendor. Moreover, the switch interfaces are not exposed to the outer world. This closed-nature of switches, delays the process of experimenting new networking architectures on the network switches [1]. Furthermore, the network operators have to manually enter the control policies into the switch infrastructure. This is a complex process of converting the high-level logics into low-level switch implementations [21]. These manual operations are error prone and therefore, attempting new innovations on the network switches is hindered, resulting in *'Internet ossification'*[22]. Businessmen must wait for a long period of time, for their ideas to be realised in the network.

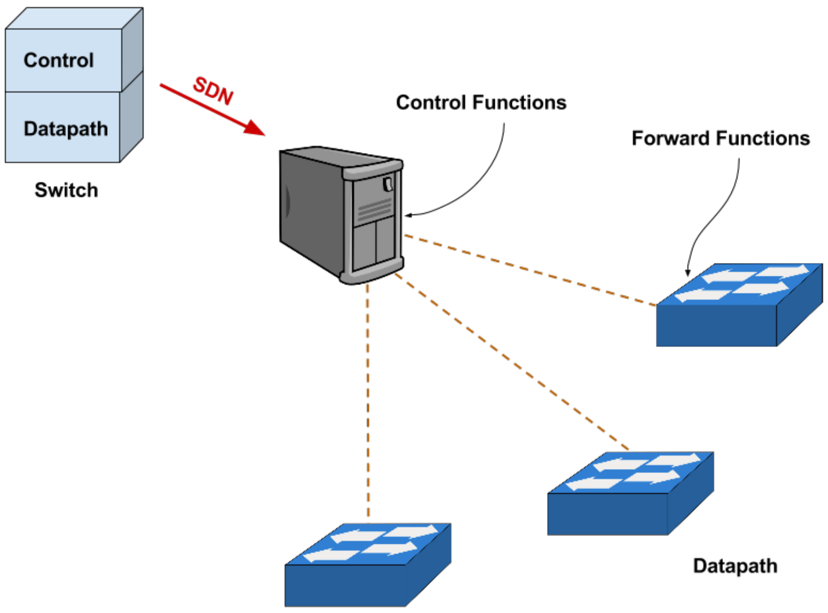


Figure 1.1: Comparison between a closed switch and SDN architecture

SDN aims to provide a solution for this problem, by decoupling the complex network control and management operations of the switches, from the forwarding operations and, forming a separate control layer [2]. Figure 1.1 shows the comparison between a closed switch and SDN architecture. Through this, SDN tries to realise a less complex data plane layer, that can be programmed by the centralised control layer. A network operator can build applications over this control layer, which use the interfaces provided by the control layer to programme the underlying switches [23]. The intelligence of the network is abstracted into this control layer and used to control the elements with the aim of getting a better quality of service out of the network. This also helps to realise the frequently changing business needs with little programming efforts instead of a whole painful hardware change or upgradations. SDN is welcomed by many industries like data centres, to have a better control over the underlying storage elements [2]. SDN helps for the continuous evolution of the networking architectures. SDN can be compared to earlier *'Active Networks'* [24] which tried providing an experimental network architecture through special packets called *'capsules'* which carry network programs along with the ordinary messages and the networks elements that process them, will execute those programs in their machines. The problems with active networks are isolation, performance and complexity. SDN has the better performance and clear separation logic compared to active networks [22].

In order to enable the control functions that are outside the switch box, to act on the switch, SDN suggests abstracting the switch knowledge into a *'substrate'* [1] which can be acted upon and programmed by the control layer. Thus, from outside the switch box, the control plane will look for of this substrate for it to successfully transfer the control policies into programs. *'OpenFlow'* [3] helps to realise this SDN switch 'substrate' in terms of *'Flow table'*. OpenFlow is a Software-Defined Networking architecture which abstracts the intelligence of the network elements into a centralised controller so that the network elements can be simple and concentrate on forwarding operations. OpenFlow also provides the interface for the switch and the controller to communicate with each other, which requires both of them to understand the OpenFlow protocol.

An OpenFlow switch is equipped with a flow table. A flow table generalises the knowledge and current state of the network switch. The flow table consists of flow entries which decide on how to handle the network flows. A flow can be any definite form of network traffic (for example, the packets that are destinated for a particular application , all HTTP packets, all packets routed to a particular country) [1]. The OpenFlow controller programs this flow table to change the behaviour of the switch. The *'TCAM'* [25] in a switch, can be utilised to implement the flow table on it. The reason for the popularity of OpenFlow is that it is being implemented by many vendors compared to other SDN implementations [26] and it provides the following major benefits [3, 22] to the network:

- Centralized control of the underlying network, that simplifies the network controlling process of the network operators
- Flow based operations, that help to realize better traffic control in the network
- Less-complex network elements, that can be programmed by the control layer
- Clear separation between the infrastructure and the control policies, so that the business ideas can be easily converted into controlling applications over the control layer.
- Load balancing for the better traffic engineering in the network
- Pro-activeness of the controller

Thus, altogether the control plane and the substrate will form an operating system for the underlying network with an abstracted view of both the control functions and the switching infrastructure [27]. The substrate in the switch focuses on executing the instructions from the controller, while the instruction programmability and the innovations are carried out in network operating system and the controlling applications [1].

1.1.2 ICN for 'Tomorrow'

In the same time, when OpenFlow was proposed and became popular in supporting IP flow-based communication, on the other side people started realising a need for a change in the Internet architecture from being *'host-centric'* to *'content-centric'* [4]. In 2006, Van Jacobson, first put forward this idea of internet architecture change in a google talk *'A new way to look at networking'* [28]. He mentioned that the entire Internet users community started realising that the internet which is originally designed to solve telecommunication needs, started facing serious performance issues recently. When the internet users care more about *'what'* they are retrieving, the internet cares about from *'where'* it is retrieving[5]. When the applications and the amount of data evolve drastically the internet architecture remained the same and started showing performance issues while trying to retrieve exabytes of data over the network from the content producers [29].

The Internet is subject to a sudden rise in the amount of data being handled which is termed as *'flash crowds'* and it is mostly due to the mobility of the devices connected to the internet [5]. *Mobile IP* [30], a patch to achieve mobility with the existing architecture of Internet, increased the complexity of the overall architecture of the Internet [29]. *'Information dissemination'* is the primary motto of today's Internet. This has become more complicated especially when the Internet has to keep up with the user experience and service agreements. The increased demand for data and the fact that the internet users are caring more about the data and less about the location of the data, are the main driving forces behind the architecture proposed by [4], which is originally termed as *'Content-Centric Networking'* and recently generalised as *'Information-Centric Networking'*(ICN) [31]. ICN results in a receiver-driven communication. The network is responsible for locating the information which is requested by the user[32]

The ICN depends on *'Named Data Objects'* [5] which can be any data *chunk* on the internet which is accessible through the network with a name. In contrary to the IP protocol, which revolves around the host addresses to provide communication between two entities to retrieve data, ICN puts forward the content which is requested by the user. Thus, it releases the constraint of getting the data only from the producer [29]. The user can get the intended data anywhere from the network and the integrity of the content is taken care by the content itself through digital signatures [4].

In ICN, all the contents are given names and they are retrieved based on these names. When a user wants to retrieve a piece of information, the user has to send out request for the information using its name and the network finds the relevant information for the user using the requested content name. The motivation behind this is that, users nowadays are interested in the content and they are not worried from where they are getting the content, but still the IP infrastructure resides on addresses. ICN can be compared with a dedicated content distribution technology like P2P [33], which is an overlay technology and content dissemination technology like CDN [34]. The network satisfies the client who requested for the content, with any copy of the content on the network, not restricting to the content from the producer [5].

ICN is gaining a lot of attention from research activities and the research interest is constantly growing on it. The reason behind the attractiveness of ICN is that the benefits which are given by it to the internet; mobility support, secured content-based communication, improved content availability and multicast communication. ICN enables the end users to be satisfied by the network layer itself. *'Load balancing'* through packets aggregation is another significant benefit of ICN [35]. ICN is best suited for today's internet to focus on efficient content distribution and mobility [5]

1.2 Project Motivation

As explained before, both OpenFlow and ICN, have their own benefits in their respective fields. The main motivation behind this study is a thought of realising an integration between these two technologies to apply benefits of one on the another. For example, flow-based communication feature of OpenFlow can be applied to ICN, to have better control over the ICN communication and, content-awareness feature of ICN can be applied to OpenFlow, to make it move towards the future internet architecture. As shown in the following 'literature synopsis' section, even though there are a number of existing works on integrating OpenFlow with ICN, they remain as the overlay approaches or do not provide a clear picture of achieving a clean state integration. We believe that there is a need for providing such a clear picture in terms of 'proof of concepts' for the integration, that can be enhanced along the line when both the technologies are matured.

1.3 Literature Synopsis

Information-Centric Networking(ICN) and *Software Defined Networking*(SDN) are two popular networking paradigms, that are predominantly considered for both the industry and the academic research activities [34]. Information-Centric Networking aims to change the internet architecture from being host-centric to content-centric by putting the contents as primary primitives in the communication, meanwhile, Software-Defined Networking helps to adapt and experiment new network technologies and protocols by separating the controlling plane from the forwarding infrastructure. Recently. there has been an increased interest among the researchers to merge both of these platforms to exploit the benefits provided by both of them [36]. Most of the studies exploited the similarities between them in order to get best of these two architectures. Even though few functionalities match between them, clearly there are many differences in the approach and the way they are implemented. Thus, it requires an immense study about both the platforms before talking about how they can be combined. Because of the fact that both the technologies are emerging in the industry, there are a lot of different solutions for integrating them [37, 38, 39, 40]. A number of studies produced non-extensions approaches, by providing an overlay of ICN on OpenFlow or, by developing a wrapper or plugin to provide ICN functionalities over OpenFlow. These studies come with a number of drawbacks which include; processing delays due to additional plugins, IP fields semantic changes due to overlay approaches and dependency on IP address instead of content names. On the other hand, a number of studies tried extending the OpenFlow to provide better support for ICN, but most of them have provided only minimal information on the possible extensions and few of them restricted their scope to either the OpenFlow switch or the OpenFlow controller. So, clearly, there is a need to study a complete extension procedure for OpenFlow in order to provide maximum support for ICN functionalities.

1.4 Project Aims

This dissertation aims to provide a solution for the question : *How to create a control plane for ICN with existing standard, such as OpenFlow, in order to realise a clean state integration between OpenFlow and ICN ?*

In particular, this study aims :

- To study the architectural differences between two emerging technologies, OpenFlow and ICN
- To focus on the research works on combining OpenFlow and ICN, particularly focusing on the research works carried out on integrating the technologies
- To give attention to the functional areas which are not covered enough in the previous research activities
- To formulate the problems or challenges behind the integration process
- To design and implement the formulated solutions and produce a modularized code base
- To evaluate the solutions by comparing with existing ICN implementations

1.5 Project Approach

This dissertation proposes 'OF-ICN' that extends the essential OpenFlow elements; OpenFlow switch, OpenFlow controller and the OpenFlow protocol, with ICN functionalities and features, in order to create an OpenFlow-based control plane for the ICN network.

This dissertation presents a design where the OpenFlow elements are equipped with ICN-related components as follows:

- ICN-enabled OpenFlow switch

    - A data structure to provide in-network caching
    - A data structure to store breadcrumbs for forwarded user requests
    - A data structure to store next-hop routes for content names
    - Ability to traverse the content name from the packet

- ICN-enabled OpenFlow controller
    - A data structure to store contents
    - A data structure to store routing details for content names

- Content producers and consumers, enabled to send packets with content names

We use open source python OpenFlow controller, POX, to implement the proposed design. We exploit the switch, controller and the protocol library modules provided by POX to realise the solution. We evaluate our solution by ensuring that the necessary ICN functionalities are provided by it and by comparing it with an existing ICN implementation, CCN [41] and its prototyping tools, CCNPing [42] and Mini-CCNx [43]

1.6 Project Contribution

This dissertation provides a complete analysis of the research works towards combining OpenFlow and ICN and, list the drawbacks and gaps that are identified in them. Instead of constructing the necessary ICN functionalities from a specific implementation of ICN, this dissertation considers the generic ICN architecture and derives the experimental functionalities from it. Also, this study presents the necessary extensions to OpenFlow, for it to handle ICN flows and, the algorithms behind these extensions. This project produces a modularized implementation using POX components to provide separate modules, that can be plugged in easily in a relevant codebase.

1.7 Terminology

This section provides a quick view on the major terminologies related to our study and they will be used in the forthcoming chapters.

- **OpenFlow :** A Software-Defined Networking approach that provides a centralised control plane layer for the underlying switching network and an interface to communicate with the switches
- **Switch :** A network element that connects devices in the network and forwards packets between them
- **OpenFlow Switch :** A switch that supports OpenFlow 'flow table' and communicates with the controller through OpenFlow protocol
- **ICN-enabled OpenFlow Switch :** A modified OpenFlow switch that supports ICN-related functionalities and features
- **Controller :** A network element that has the global view of the underlying network of switches and, communicates the controlling decisions to it
- **OpenFlow Controller :** An OpenFlow element that forms the control layer for the underlying network of OpenFlow switches
- **ICN-enabled OpenFlow Controller :** A modified OpenFlow controller which supports ICN-related functionalities and features
- **OpenFlow protocol :** An interface for the OpenFlow switch and the OpenFlow controller to communicate with each other. It is a set of messages, actions and events which the OpenFlow switch and the OpenFlow controller should obey
- **OpenFlow Message :** OpenFlow messages are agreed OpenFlow protocol messages, which the OpenFlow switch and the OpenFlow controller can use to communicate with each other

- **Interest packet :** A user request packet to retrieve a content that contains the name of the content. This is an ICN packet
- **Data packet :** A packet that is sent in response to an interest packet. This is an ICN packet that carries the content
- **Flow table :** An OpenFlow substrate of the switch that contains the forwarding rules for the switch. It contains a list of flow entries
- **Flow entry :** An entry in OpenFlow flow table which contains three fields: Match, Action, Counter
- **Match :** A field in the flow table which is a set of header fields against which a packet's header fields can be compared with
- **Matching :** A process of comparing the packet's header fields with the flow table entries(with each flow entry's 'match' field)
- **Action :** A field in the flow table that instructs the switch on what to do with a matched packet
- **Counter :** A field in the flow table that increments for every packet that matches with a flow entry
- **ICN :** An Internet architecture which identify 'Named data objects' in the network using the content names, instead of addressing the end-point
- **Named data object :** Any chunk of data in the network which can be identified by a name identifier
- **Cache :** Internal memory that is used for storage
- **Routing database :** The database that stores routing information for different content names

- **Content Store :** In short, it is called *'CS'*. 'Content Store' is an inbuilt cache exploited by a switch to store named contents
- **Pending Interest Table :** In short, it is called *'PIT'*. 'Pending Interest Table' is an ICN table structure which is used by the ICN-enabled switch to store the interest packets which are sent upstream by it and pending to receive a data packet
- **Forwarding Information Base :** In short, it is called *'FIB'*. FIB is similar to OpenFlow's flow table. FIB is an ICN data structure that stores next-hop route information for content names
- **Face :** A face is an abstraction of an interface through which the ICN switch can receive and send packets. A face can connect with an application, a process, a device or the Internet
- **POX :** An OpenFlow controller, which provides the software version of essential OpenFlow elements(switch, controller, library) for prototyping and experimentation
- **Event :** An event triggers based on the OpenFlow message received by the POX components
- **Event handler:** A piece of code that handles an event
- **Southbound interface :** In SDN and OpenFlow jargon, a 'southbound interface' is the interface through which the controller connects to a switch. OpenFlow protocol is a southbound interface
- **Host :** Any end-user device that connects with a switch

Chapter 2

State of Art

This chapter analyses the existing works that are carried out in the areas concerned by this dissertation. On a top level view, the literature related to this study is divided into six major sections as follows:

1. **Information-Centric Network(ICN) :** This section outlines the major ICN implementations, ICN-based tools, ICN-based routing protocol and a quick view of the interest and data processing

2. **OpenFlow, an SDN approach :** This section talks about the major OpenFlow software-switch implementations, major OpenFlow controller implementations and a number of OpenFlow-based tools

3. **Functional similarities and variations :** This section presents the major features between ICN and OpenFlow that are similar and the features which are different from each other

4. **Initiatives in industry :** This section lists some recent industrial projects that are working towards integration between OpenFlow and ICN

5. **Non-extension-based approaches :** This section analyses the OpenFlow-ICN integration approaches, that tried realising the integration without changing the

OpenFlow components and their implementation features. This section ends with a summary of these non-extension approaches and their drawbacks

6. **Extension-based approaches :** This section analyses the approaches, that tried extending or modifying the OpenFlow protocol in order to enable it with ICN functionalities. This section ends with a summary of these extension approaches and a discussion on it

2.1 On Information-Centric Network

Information-Centric Networking shifts the current internet architecture from dependency on IP addresses to dependency on named contents [4]. That said, all the contents (each and every chuck of the content) are named in the ICN world. The content name can be hierarchical, containing the named components to denote different levels of accessible data or, it can be a flat name, identifying a single file [34]. Once the contents are named, a user who likes to access a content anywhere in the network has to express an interest towards the content. This interest will pass through the network hop by hop until it reaches a copy of the content. Once the content is identified, it will be sent back to the user who requested the content. There are many implementation of ICN including; NDN [44], CCNx [45], and CONET [46]. These major implementations are outlined here:

2.1.1 NDN

Internet Protocol(IP) is designed to solve telecommunication's problems [47]. NDN is an ICN approach, which is designed with the objective of solving the problems with IP, while handling exabytes of data. NDN architecture works based on the 'named data objects' and identify them on the network by their names. NDN can also be used as an overlay over IP or it can entirely replace IP. Thus, it is called an *'universal overlay'* to IP [44]. According to Zhang et al(2010), NDN architecture suggests separating the routing functions from forwarding functions and, enable continuous research on the routing protocol, while the underlying switching network can be equipped with NDN-related forwarding operations. NDN is not restricting the network applications with a specific naming scheme. The applications can follow any naming scheme that is suitable for them. One of the objectives of NDN is to improve the 'reception rate' of the destination by improving the amount of successful request deliveries to the destination. NDN is forked from PARC's CCN implementation, CCNx, and it added new features and language supports [31]

2.1.2 NFD

'NDN Forwarding Daemon'(NFD) is based on the NDN approach and follows the NDN protocol. Recently, the entire code of CCNx is moved to NFD and additional features are added to it. One such improvement is to support TLV(*Type-Length-Value*) format for the packet structures. Now, NFD has become the core component of NDN [48]. It is a *'forwarder'* daemon that reads an interest packet and, finds out the next-hop for the packet and then, forwards the packet towards the content [49]

2.1.3 NLSR

'Named-data Linking State Routing(NLSR)' protocol is a primary routing protocol for NDN [50]. It provides the routing information based on the content name prefixes. It is achieved by NLSR, by providing a ranking for the forwarding paths which is an enhancement to NDN's *'adapative forwarding strategy'* [29]. The variation of OSPF routing protocol, OSPFN [51] is initially utilised for providing routing information for NDN network. NLSR is developed later and it differs from other routing protocols, in that it uses NDN's interest and data packets to send routing updates, whereas, OSPFN identifies routers based on IP address. The other major difference is that, NLSR can provide multiple next-hop routes for a router using a 'Dijkstra algorithm' [52] and for a content name using prefix LSA. In contrary, OSPFN can provide only a single next-hop route for a content name. NLSR helps the NDN nodes to build the topology of the network and to disseminate routing packets into the network.

2.1.4 CCNx

In 2009, Jacobson together with *'Palo Alto Research Center'* (PARC) announced an implementation for Content-Centric Networking(CCN) architecture under the project, CCNx [4]. The NDN and CCNx architecture are functionally equivalent as NDN is actually a fork from CCNx project. CCNx was originally designed to be utilised in data centres and small sensor networks. Like NFD, CCNx also supports TLV format for encoding the messages to send them on the wire [45]. CCNx names the data as 'content objects' and identify it using a 'Labeled Content Identifier (LCI)' [45].

2.1.5 CONET

[46] CONET provides an integration approach to realise ICN architecture, in which, IP layer is extended to include content-based information in it. To achieve this, *'IP option'* field in an IP packet is exploited to carry content-related information(for example, content identifier). CONET solution for ICN works between different sub-networks and the CONET border nodes are kept at each sub-network to handle CONET packets. The border nodes forward the packets based on the content names contained in the packets. Thus, only the border nodes act upon the content name and route the packet towards the next border node close to the content copy. Within each sub-network, the routing is carried out based on the type of the network(eg., IP). CONET architecture does not provide bi-directional communication, as it does not store state information in the network elements along the path of the requests. Instead, the architecture suggests adding CONET based packet traversal addresses to the packet itself. The data packet uses this information in the interest packet to traverse back to the requester. CONET also provides a *'Name Routing System'*(NRS) [12] to look up for an unrecognised packet during the

routing process.

2.1.6 An ICN node

Despite many implementations of ICN concepts, the basic working culture is same in all the implementations [34]. This dissertation studies the general ICN architecture based on the common features for all the different ICN implementations. In order to achieve an ICN communication, two packets are involved [4]. One is called *'Interest'*, which expresses the need of the requester. The requester can be a machine on the same network or an application process running on the same machine. The interest packet contains the name of the content to be retrieved. This interest packet traverses the network towards the content. The other packet is *'Data'*, which is the actual content response for the interest. A data packet contains the name of the content, the actual data and the security signatures. The data packet traverses in the reverse path of the interest in the network until it reaches the user, who requested the content. Intermediate nodes that forward the ICN packets are termed as ICN nodes and each ICN node use three data structures to do interest forwarding : *Forwarding Information Base(FIB), Pending Interest Table(PIT) and Content Store(CS)* [41]. 'FIB' is the table, that contains the next hop information for a list of contents. 'PIT' is the table, that contains information about the interests that are forwarded upstream by the node towards corresponding data. 'CS' is a table, that stores the content received by the node and forwarded towards the user. The interfaces through which the interests are received by the ICN node are called *'Faces'* [5]. Figure 2.1 shows the architecture of an ICN node.

2.1.7 Expressing the interest

Once an ICN node receives an 'Interest' through a face, the node extracts the content name(prefix) from the packet and, searches in the CS, for a copy of the data by performing *'longest prefix match'* [39]. If it matches with a CS entry, the corresponding content is

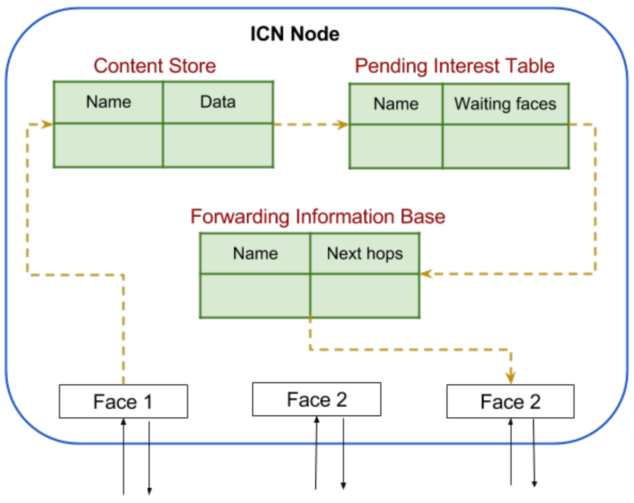


Figure 2.1: An ICN node and its elements

encapsulated in a 'Data' packet and sent back in the interface through which the interest is received. If there is no match, then the node checks for a match in PIT. If an entry matches in PIT, which means an interest has already sent for the content. The current interface is added to the list of interfaces which are waiting for the same data. If there is no match in PIT, then the node checks for a match in FIB table. If an entry matches with FIB, then the corresponding next hop face is used for forwarding the interest packet. The node puts the interest packet in the corresponding face. Each ICN node, that is on the path towards the content will process the interest in the same way. After forwarding the interest, the node adds an entry in PIT with the requested prefix and the interface which is waiting for the content. By this, the interest packets leave *'breadcrumbs'* on the path they are travelling in the network [53].

2.1.8 Bi-directional communication

In ICN, not only the producers of the content, but also the intermediate network elements store the content [29]. Each ICN node forwards the interest packet so that it reaches either the producer of the content or an intermediate node, that has stored a copy of the content.

Once the content or a copy of the content is reached, the content is encapsulated in a data packet and sent back to the requester in the reverse path following the breadcrumbs (PIT entries) left by the interest in the intermediate ICN nodes. Each intermediate ICN node, when it receives a data packet, checks for the PIT entry. If there is a match with PIT entry, it puts the data packet in the waiting interfaces. After putting the data in the interfaces, it deletes the PIT entry and thus, the content *satisfies* the interest. If no PIT entry exists for a data, the data is dropped assuming that the interest is expired before the data arrival. In addition, each ICN node on the reverse path stores that content in its cache. Thus, further interests for the same content do not travel all the way to the producer, but any nearest ICN node which cached the content satisfies the interest by sending back the cached content [28].

2.1.9 Tools

This section describes the background of some of the recent ICN prototyping tools.

- **Mini-CCNx :** 'Mini-CCNx' [43] is a prototyping tool based on 'Mininet' [54], to realize CCNx implementation. The main motive of Mini-CCNx is to provide flexibility and exactness(fidelity) to ICN experiments. In Mini-CCNx, ICN nodes are connected through *'virtual Ethernet links'* but in a single machine, which can be a laptop. It works on real CCNx code, that overlays TCP or UDP connections. If CCNx code is updated, Mini-CCNx will automatically use the new code. Thus it provides smooth integration to the real network environment, because of which, it comes under the category of *'emulator'*. Mini-CCNx provides both the options of a configuration file and GUI to create topologies for experiments. Using Mini-CCNx, hundreds of various ICN topologies can be built on a single machine. CCNx code allows Mini-CCNx to use 'ccnd' [55] daemon to enable FIB, PIT and CS in the node. Like Mininet, Mini-CCNx isolates the experiment from the network connections through virtualisation approach. The notable difference between Mininet and Mini-

CCNx is that, Mini-CCNx connects virtual ICN nodes in a point to point fashion instead of utilising switching elements [56]

- **Mini-NDN :**

  Like Mini-CCNx, 'Mini-NDN' [57] is also based on 'Mininet' architecture. It is an emulator helps to realise ICN implementation in terms of 'NDN' architecture, unlike Mini-CCNx, which works on 'CCNx' architecture. Mini-NDN runs 'NFD' and 'NLSR' code on top of virtualized nodes, to provide a prototype of NDN architecture. Mini-NDN tries to provide an emulation environment that is close to real network environment [49]. This tool is developed out of a strong need for a testing environment during the development of NLSR and, extended from Mini-CCNx. Mini-NDN also provides two ways to define the required topology. One is the configuration file and another one is GUI component. Using either of the options different network configuration parameters like bandwidth, delay and packet loss rate can be configured for the topology to be used [31]. The Mini-NDN code is available as an open source code in GitHub [57].

2.2 On OpenFlow

OpenFlow is a pioneer in realising SDN concept, by defining the specification for the switches with new features and interfaces, to improve the programmability in them [3]. The main aim behind OpenFlow is to support experimentation of new addressing and routing protocols as well as new network architectures [1]. OpenFlow does this by exploiting the flow tables of the switches and by controlling them using a specialised control plane layer called *'controller'* [26]. Thus, the main components involved in OpenFlow are 'OpenFlow switch' with flow tables, 'secure channel'(SSL) to connect with a controller and, the 'OpenFlow protocol' to define standard message structures between the switch and the controller. An OpenFlow 'flow table' has match fields (tuples) to match against the incoming packet, an ’action set’ to act upon the matched packet and ’counters’ to keep track of the number of packets being processed. The switch can communicate with the controller if it is not able to find a flow entry for a packet, by sending a part or the complete packet(as per the configuration) to the controller. The controller will decide on how to forward the packet and push down a flow rule which is saved in the switch’s flow table. Then the packet will be sent back to the switch and processed by it, based on the rules sent by the controller [3]. Figure 2.2 shows the comparison between a legacy switch and an OpenFlow-enabled switch.

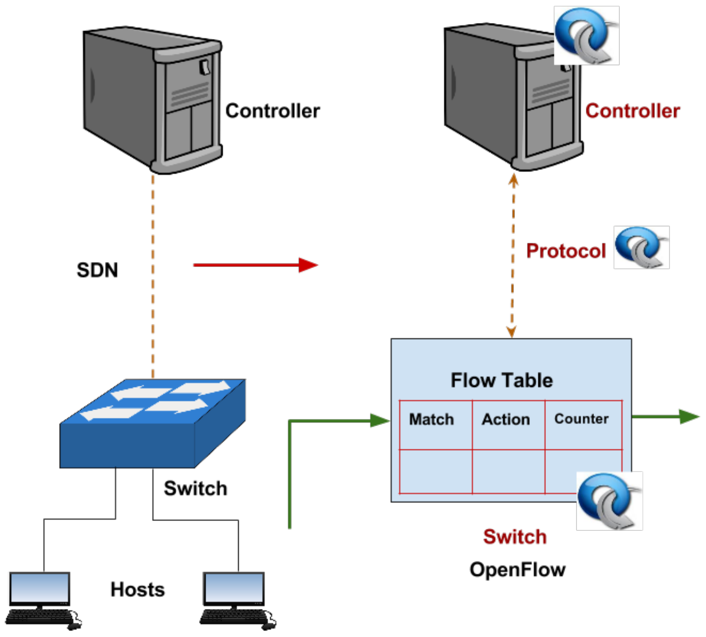


Figure 2.2: Relation between SDN and OpenFlow [1]

2.2.1 OpenFlow software switches

There are a number of software implementations of OpenFlow switch specifications including reference switches and virtual switches. The major ones are quoted below :

Open vSwitch is a widely used Linux-based multilayer SDN switch, that comes with the OpenFlow support [58]. Unlike usual network switches, Open vSwitch supports virtual machines and works over the hypervisors running in those virtual machines. It comes with a daemon that runs over the *'kernel datapath'* from which it receives the packets and match against the flow table. The daemon communicates with the controller using OpenFlow protocol. Open vSwitch comes with the support for both the kernel and user space switching operations.

CPqD [59] is a switch, which is a tweaked version of Ericsson's software switch. When many of the hardware switches support OpenFlow 1.0, in order to increase the research activities and innovations using latest OpenFlow specifications, CPqD software switch is built upon OpenFlow version 1.3 and this switch is generally called *'OpenFlow 1.3 software switch'*. This switch comes with the following software components; datapath, OpenFlow protocol library, secure channel to connect with controller and a console-based tool to configure the switch

2.2.2 Controller implementations

'Controller', in general, is one of the main components on SDN, which handles and operates on the abstracted view of the network topology [60]. Controller resides in the control plane or the control layer and, directs the working nature of the underlying network infrastructure. The SDN controller is exploited by the network administrator or the network operator to change the controlling aspects of the network or even program the infrastructure mechanism [2]. The OpenFlow controller is an SDN controller, that guides the switches, that are connected to it and, makes decisions for the switches to forward packets in the network. Being an SDN controller, OpenFlow controller has the view of the entire network that is being connected to it and, it maintains the abstracted statistical and capability information of the network elements that are connected to the network [39]. Using this information, the OpenFlow controller will make the forwarding resolutions for the switch that requested for it. The controller can reside on the same machine where the switch is running or it can run on a separate machine [21]. The controller can be a centralised one in which all the information about the network will be stored in that centralised controller and the decision-making process will be by the same controller. On the other side, the controller can be a decentralised one, in which the control plane is divided into a number of machines which store the distributed information about the switch and the forwarding decisions will be taken in a collaborative way involving all the necessary controllers in the decision-making process [2]. In either way, the OpenFlow switch and the OpenFlow controller are connected through the 'OpenFlow protocol', that specifies the messages that can be passed between the switch and the controller. The OpenFlow controller is expected to maintain a routing table whose subset will be stored in the flow table of the switches [26]. A switch will contact the controller whenever it is not able to decide on how to forward a packet and the controller will use the routing table to identify a destination for the packet and, push a flow entry down to the switches, for them to process the packet and then forward it to the corresponding destination.

Software-Defined Networking outlines the need for the network operating system to control and manage the underlying network using a high-level abstracted view of the network, instead of low-level control actions [27]. These networking operating systems are expected to provide a programming interface through which the underlying network elements can be managed and controlled. The network applications can be built over these operating systems and the applications exploit the abstracted view maintained by the network operating system and make controlling decisions [23]. The network operating system takes the responsibility of communicating the control decisions to the network elements through the programming interface. These network operating systems are the controllers in general and the programming interface provided by them, is the protocol [61].

A lot of implementations of OpenFlow controller are available, that vary in the language, supported OpenFlow version and the performance. Few details about some of the OpenFlow controllers are listed below:

- **NOX** is an OpenFlow controller implemented using C++ language and, it was developed at the same time when the OpenFlow is developed. NOX is seen as the first OpenFlow controller. Nox supports OpenFlow 1.0 and it comes with a set of applications to create the abstracted view of the underlying switches, to insert, delete and modify entries in the switches' flow tables and to invoke events based on the changes in the underlying network [61] .

- **POX** is an OpenFlow controller derived from NOX and developed in Python language. POX controller aims at providing an easy and flexible environment for carrying out SDN experiments and research activities. POX works on the component-based model, where all the network elements and activities are realised as individual components, that can be separated and used wherever the need is. Like NOX, POX also helps to develop network management applications over the programming interface, to make controlling decisions for the network under control. POX supports OpenFlow 1.0 [62] .

- There are a number of other OpenFlow-based controllers including; Ryu [63], FloodLight [64] and OpenDayLight [65]. Like POX, **Ryu** is a component-based OpenFlow controller which supports a number of OpenFlow versions; 1.0,1.2,1.3,1.4 and 1.5. Ryu also supports multiple programming languages to create network control applications. **Floodlight** is a Java-based OpenFlow controller, that uses JAVA or REST API for communications and expresses network applications as services. **OpenDayLight** is a multi-vendor SDN controller, that aims in controlling the *interoperability* between the SDN implementations [60] .

2.2.3 Tools

- **Mininet :** Mininet [54] is a fast prototyping tool to realise Software-Defined Networking architecture. Unlike a simulator, which is not close to real world environment, Mininet achieves this closeness to the real world by using the real code, that will be used in production base and builds the network on top of it. Mininet operates on the basis of virtualizing the underlying operating system into multiple processes and 'namespaces', which allows it to provide support for hundreds and thousands of switches in a single laptop. Mininet aims to provide an easy prototyping platform for new ideas including new network protocols, modification to the protocols and even new network architecture [66]. OpenFlow, being a major realisation of SDN, Mininet provides complete support for various OpenFlow versions and switch types. For example, the user can either select a kernel-space switch or an user-space switch based on the requirement. It also comes with the support for a number of OpenFlow controller including, NOX and POX. Mininet switches can be connected to a local controller or to a remote controller. It provides interactive tools for the users to directly interact and configure the network prototype under experimentation

2.3 On functional similarities and variations

This section covers the functional similarities and variations between OpenFlow and ICN architectures.

2.3.1 On packet format

OpenFlow is mainly designed to operate on IP-based packets [3]. It also supports other transport layer protocol packets like TCP and UDP. Current OpenFlow switches in the market are equipped to support IP packets [17]. When they receive a packet, the packet headers will be parsed and matched against the flow table entries. IP packets contain various header fields like source address, destination address, Type of Service(TOS) and IP options. OpenFlow supports almost all of these header fields in addition to the Ethernet, TCP and UDP fields. Starting with the support for a subset of packet header fields, OpenFlow (as per the current specification OpenFlow 1.5.0) now supports almost all the fields in the packet header including IPv6 fields [67]

The packet format is important because, it decides on how the packet is going to be forwarded by the switch. A lot of switches drop the packet when they cannot recognise the format of the packet and, there is no rule for them to forward the unrecognised packet to a controller [3]. The term *'unrecognised'* here can denote a packet, that does not match any of the rules in the switch or, the packet with a format which the switch does not understand. The later case is subjective and differs in different switch configurations and working methodologies. When the IP packets contain details about end-point addresses, ICN packets contain details about the data or the content that is requested [4].

There are many ways to express the content name which will be discussed in the following sections. The overall point that has to be considered here is that the 'content naming' in ICN is still under research and consideration and, there is no solid finalisation on how the 'content name' should be expressed in an interest packet [29]. That said, the packet format which depends on the content name will also change for different naming methodologies.

As already mentioned, OpenFlow has no support for ICN packets yet [67]. In addition, there is no easy way to decide on what ICN packet format OpenFlow should support when the ICN packet format itself not yet standardised. There are a number of workaround solutions in the industry for the OpenFlow to recognise content names (in general ICN specific packets):

- Hashing the content name and including them in one of the OpenFlow-recognised fields like IP destination or transport port fields [10]

- Converting the content name into an IP address so that it can be included in the IP fields [68]

- Using the IP option field to carry ICN related information (Note : Most of the OpenFlow specification does not read IP options) [37]

- Adding a tag to the IP packet with the ICN content name information, that can be untagged at egress switch [69]

2.3.2 On matching

'Packet matching' is the process, that decides, what will happen to a packet after reaching the switch. OpenFlow switches are equipped with flow tables in order to do this process [26]. A flow table is a container for flow entries where each entry contains a rule, instructions and counters. A *'rule'* is a set of header fields, that the switch should match against an incoming packet. The header fields can be wild-carded in a rule for the all the packets to match with that rule. The switch will parse the incoming packet, extract the packet headers and match against the corresponding fields in the flow table. When a packet matches a corresponding flow table entry, the instructions associated with that flow entry will be executed. The instruction can be an action to send the packet to next table for processing or, it can be the one asking the switch to drop a packet or, it can be the one asking the switch to send the packet to the controller or, it can be any other actions. The flow entries are created or pushed to a switch by the controller which is giving the responsibility for the control plane to decide on how the switch should act on a packet. The counters in the flow entries are for *statistical* purposes. When a packet matches a flow entry, corresponding counters will be incremented in order to keep track of the statistical information [21]. This statistical information can be used by a control layer to make controlling decisions.

The matching process is important as it decides how the switch is going to operate and, it gives the provision for abstracting the intelligence from the switch and providing them as mere rules for them to follow upon. In ICN, 'matching' is a *'pipeline'* process, where the various ICN-based data structures(FIB,PIT,CS) are utilised and the content prefix is matchched with the tables for a *'longest prefix match'* [35]

2.3.3 On forwarding

The main aim of Software-Defined Networking(SDN) is to separate the forwarding infrastructure and routing intelligence so that the network elements like switches will take care of forwarding process alone and the control plane can take the responsibility of guiding the switch with the forwarding and routing decisions [2]. In a simplified way, the forwarding processing in a switch is to send out a packet out to a particular port or interface. As mentioned in the previous section, this forwarding process purely depends on the matching process. Mostly the instructions in the flow entries are to forward the matched packet to a particular destination. OpenFlow switches are by default with some forwarding rules (depends on the vendor) and further flow entries are pushed down by the controller as the switching process takes place. OpenFlow switches can forward a packet in different ports, that can be physical ports or virtual ports [70]. There are many virtual ports supported by OpenFlow. For example, *OFPP_CONTROLLER* is a virtual port, that asks the switch to send the packet to the controller and, *OFPP_INPORT* is a virtual port, that guides the switch to send the packet back in the incoming port. There are many such virtual ports available in the OpenFlow specification. As far the OpenFlow is considered, there is no differentiation between the packet which queries a destination or the packets that carry some data towards the requester [3]. All the packets carry a source address and destination address and OpenFlow purely matches based on them and other fields that are used for matching. Thus the response from the destination also will be treated as a normal packet and matched upon the flow table. There are no packet 'breadcrumbs' left for the OpenFlow to decide where to the send the response for a request packet, that is sent upstream. This semantic is opposite to ICN methodology, in which the response packets are aligned with the interest/query packets and, the switches are expected to store some breadcrumbs for the response to reach back the requester [4]. As far as OpenFlow forwarding is considered, flow table is the major structure used for deciding on the forwarding action. The controller is expected to have the overall routing table from which the controller decides when a switch is not able to find a matching entry. In ICN, 'Forwarding Information Base' plays a major role in deciding where to the packet next. This table holds the content names, for the packets to be matched upon and the next-hop route to send the corresponding packet towards the content copy [29].

2.3.4 On caching

Caching is a process of storing some information in the available buffer capacity in a network element . OpenFlow switches are endowed with the capability to cache the flow entries [21]. OpenFlow specification does not specify caching any details other than the flow entries, assuming the fact that the buffer size of the switches will be less. The buffer is a small memory that comes along with the network elements to store some processing information which can be temporary or permanent. OpenFlow exploits this buffer to store forwarding rules in the switch. The buffer size differs between different switches and between different vendors. There are some switches which are available in the market that has a lot of inbuilt buffer memory which can be exploited. OpenFlow switches will be able to store flow entries that can be accommodated within the allocated buffer memory. If the flow table reaches its size then the controller has to take actions to either drop some entries or replace some entries based on some algorithm [71]. *'In-network caching'* is one of the key functionality of ICN, which allows the network elements to store some content in their memory in order to send the response to the requester as quickly as possible instead of sending the query all the way to the destination as happens in IP forwarding [72]. ICN architecture suggests the network elements to store popular content when they forward a content for the first time so that later interests or requests are satisfied by them.

2.4 On initiatives from industry to integrate OpenFlow and ICN

This section outlines two European projects that recently focus more on realising ICN functionalities over OpenFlow.

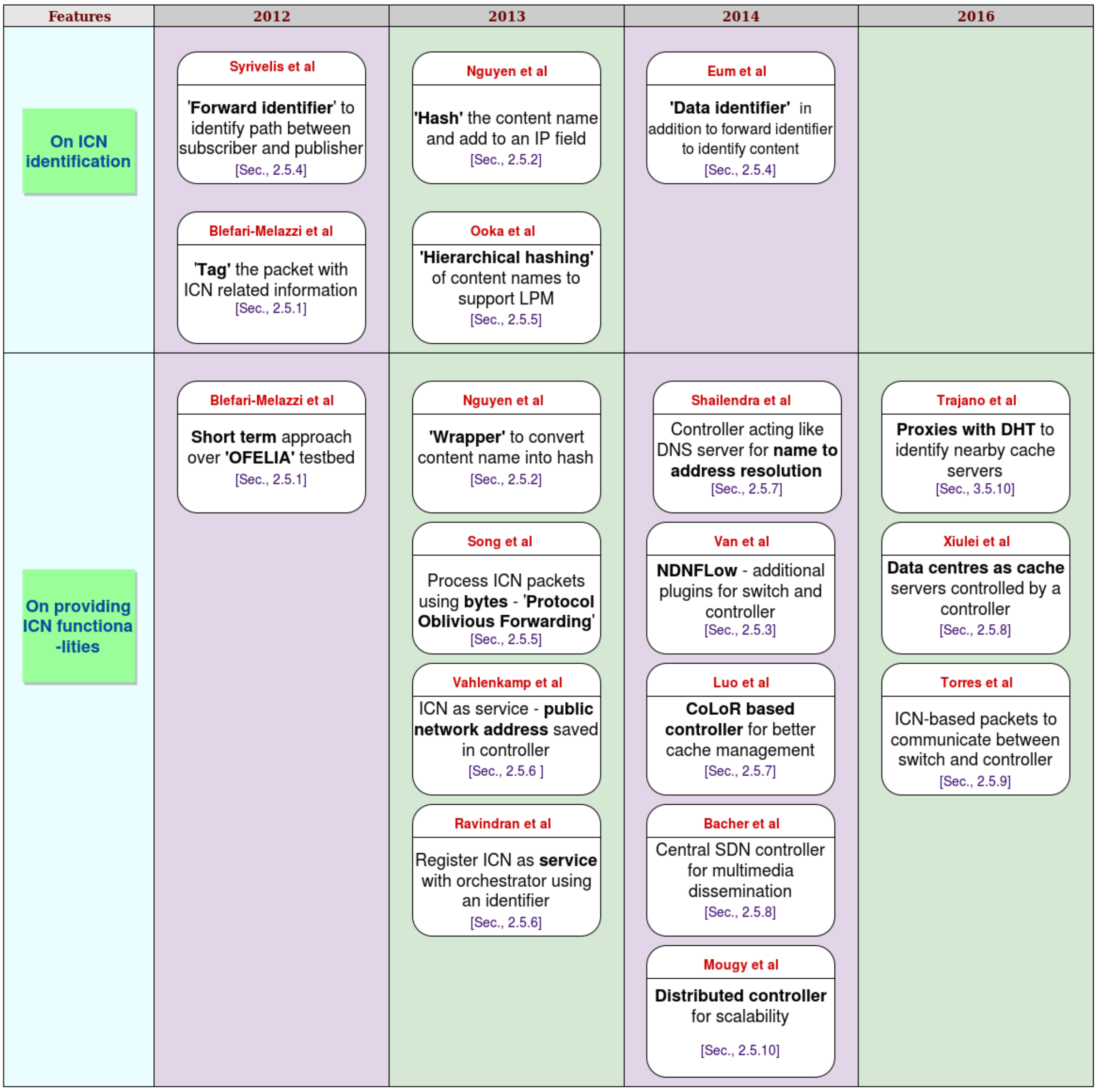
2.4.1 OFELIA

'OpenFlow in Europe - Linking Infrastructure and Applications'(OFELIA) is a part of 7th framework programme European project [73]. This is the framework used for real network evaluations of the experiments on controlling the underlying network and, this is operational from August 2011. It provides islands of various networks, for realising different experiments on them. For example, *'CNIT'* is an island, that is focusing on Information-centric networking experiments [74]. These islands provide virtualised end hosts, that act as data plane for the OpenFlow controller which decides on forwarding rules. OFELIA provides the control framework for the experimenters to register with the facility and, provides a UI for them to run experiments. The basic idea behind OFELIA is to control the routing of ICN interest and data packets and efficiently realise in-network caching using SDN standards [37]. CONET is experimented over OFELIA by mapping the content names into fixed length tags, that are transported in UDP source and destination ports (because OpenFlow cannot read IP options) [74]. There are two directions of work being carried out in OFELIA to enable ICN communications over SDN and they are named as *'Short term'* and *'Long term'*. These approaches are outlined in the sections : 2.5.1 and 2.6.2. The project deliverables from OFELIA are named under the term 'EXOTIC'[75].

2.4.2 ALIEN

'Abstraction Layer for Implementation of Extensions in Programmable Networks'(ALIEN) [76] is a major research activity on developing a *'Hardware Abstraction Layer'*(HAL), that allows the non-OpenFlow compatible network elements to be placed in OpenFlow control framework. [77] evaluated the ALIEN HAL by integrating it with OFELIA's ICN implementation on using CONET architecture over OpenFlow framework.

2.5 On non-extension approaches



Features	2012	2013	2014	2016
On ICN identification	Syrivelis et al: **'Forward identifier'** to identify path between subscriber and publisher [Sec., 2.5.4] Blefari-Melazzi et al: **'Tag'** the packet with ICN related information [Sec., 2.5.1]	Nguyen et al: **'Hash'** the content name and add to an IP field [Sec., 2.5.2] Ooka et al: **'Hierarchical hashing'** of content names to support LPM [Sec., 2.5.5]	Eum et al: **'Data identifier'** in addition to forward identifier to identify content [Sec., 2.5.4]	
On providing ICN functiona -lities	Blefari-Melazzi et al: **Short term** approach over **'OFELIA'** testbed [Sec., 2.5.1]	Nguyen et al: **'Wrapper'** to convert content name into hash [Sec., 2.5.2] Song et al: Process ICN packets using **bytes** - **'Protocol Oblivious Forwarding'** [Sec., 2.5.5] Vahlenkamp et al: ICN as service - **public network address** saved in controller [Sec., 2.5.6] Ravindran et al: Register ICN as **service** with orchestrator using an identifier [Sec., 2.5.6]	Shailendra et al: Controller acting like DNS server for **name to address resolution** [Sec., 2.5.7] Van et al: **NDNFLow** - additional plugins for switch and controller [Sec., 2.5.3] Luo et al: **CoLoR based controller** for better cache management [Sec., 2.5.7] Bacher et al: Central SDN controller for multimedia dissemination [Sec., 2.5.8] Mougy et al: **Distributed controller** for scalability [Sec., 2.5.10]	Trajano et al: **Proxies with DHT** to identify nearby cache servers [Sec., 3.5.10] Xiulei et al: **Data centres as cache** servers controlled by a controller [Sec., 2.5.8] Torres et al: ICN-based packets to communicate between switch and controller [Sec., 2.5.9]

Figure 2.3: Non-extension approaches on OpenFlow and ICN integration - Timeline map from 2012 to 2016

Two different approaches are considered in literature to enable OpenFlow to support ICN functionalities. One is to overcome the limitations of OpenFlow by suggesting additions and extensions to build new specifications, supporting new features. The other one is to use legacy OpenFlow elements and, try to work around it, to make it support

ICN functionalities. Former approach will be discussed in next section. In this section, let us take a look at the research activities on the latter approach, that tried to achieve ICN communication in OpenFlow without modifying it. This section will discuss the advantages and disadvantages of these solutions on reaching the futuristic goal of making OpenFlow support ICN. We believe that discussing the pros and cons of non-extension approaches will provide the need and motivation for the experiments on the OpenFlow extension approach. Figure 2.3 shows a summary of these approaches in year based timeline along with section numbers.

2.5.1 Short term OFELIA approach

In 2012, Blefari-Melazzi et al, suggested a *'short-term'* solution to implement ICN in OFELIA testbed [37] [73]. It is an integration approach to support one of the most popular ICN implementations, CONET architecture in SDN field [46]. The CONET architecture suggests adding an 'IP option' in normal IP packets, with the ICN information so that they can be used to search content in the network. Most of the OpenFlow standard implementations are not capable of parsing IP options. In order to overcome this weakness of OpenFlow and still to support ICN communication, the author suggested the border nodes at each CONET subsystem to add a tag to the packet based on the ICN information contained in the packet. It is suggested to insert the tag as one of the packet fields which is supported by OpenFlow so that OpenFlow network can act on the packet [37]. This solution changes the semantics of the IP fields and it requires *'tagging'* and *'un-tagging'* at each level, which is a costly process. It assumes that the underlying subsystems are IP based subsystems, as the border nodes still work with IP addresses, which may not be a possible solution for ICN clean state implementation.

2.5.2 Wrapper

A similar solution is provided by Nguyen et al., who proposed to hash the content names into an integer and, replace it in any of the IP fields in the packet, which OpenFlow can handle. This conversion is carried out by a separate module called *'wrapper'* which reads the ICN interests the switch receives, hash the content prefix into an IP field in the packet and sends back the packet to the switch so that OpenFlow can process it [78][10]. This solution avoided the need for changing the IP protocol itself unlike previous CONET based solution, that suggested adding an IP option to store ICN values, but it still suffers from the problem of changing the semantics of the existing IP field by replacing it with ICN based details.

2.5.3 SDN-NDNFlow

Like the 'wrapper', in [11], a solution called *'NDNFLow'* is introduced with additional plugins for both switch and controller in order to operate on the interest packets. 'NDNFlow' operates on a *second application specific layer* parallel to OpenFlow. A separate communication channel and a controller module are involved in the already existing OpenFlow communication channel and process. The controller is wrapped with an additional ICN module and the Open vSwitches are enabled with ICN capabilities through one of the ICN implementations, CCN and its CCNx daemon. In this solution, switches need to make additional connections and handle different message protocols. Adding an extra module over the controller to handle ICN module may result in conflicting decisions between the modules and, result in a processing delay.

2.5.4 Fixed identifiers

In the same year, Syrivelis et al., suggested using unique 'fixed sized labels' called *'forward identifiers'*, that are actually LIPSIN identifiers, that work with *'publish-subscribe'* systems [38]. These identifiers identify a forwarding path between a subscriber and a publisher, instead of identifying the content. This identification is not aligning with the general ICN architecture. The controller is flooded with this solution in order to assign or resolve forwarding identifiers in the packets, based on the forwarding path for that information retrieval. Even though it helps to retrieve information over the network, it still identifies a path to the producer, instead of the content, which falls behind supporting efficient in-network caching. In a similar way, but to identify the content as well, in [79], an additional identifier called *'data identifier'*, is introduced which is a fixed length identifier to identify the data that has to be retrieved. Both the forwarding and data identifiers will be put in the IP packet fields, which the OpenFlow can understand. Again, this changes the semantics of the IP fields and also involves costly encoding and decoding processes for those identifiers.

2.5.5 Hierarchical hashing

All the above solutions suggested to use fixed length flat hash values or identifiers to identify a particular content, thus, resulting in a large number of individual identifiers to identify a large number of contents in the network. In [80], a *'hierarchical hash value'* is suggested instead of a flat hash value in order to provide *'Longest Prefix Match'* for the content names and, to enable content name prefix 'aggregation' in the forwarding tables. An alternative to hashed names is suggested in [81] using *'Protocol Oblivious Forwarding'*(POF) [82], which eliminates name-to-hash tag mapping by processing the content names using the bytes in the packet.

2.5.6 Service-based overlay approaches

There are a number of overlay solutions in which ICN functionalities are implemented over existing IP networks, to realise and experiment ICN in a short term, with the available technologies and legacy systems. In [68], existing ICN networks on the internet are considered as separate domains, that provide ICN services with *'public network addresses'*.

A controller, with global knowledge of such domains, is suggested. The user who is trying to retrieve a content from an ICN domain should send an interest to the switch and the switch forwards the packet to the controller. Controller decides on the ICN domain responsible for the content and returns the public network address of that domain to the switch. Then the communication takes place between the switch and the ICN domain. Similarly in [83], Ravindran et al., realised ICN as a service, wherein, content producers register the service with the *orchestrator*, which can be an SDN controller. The users access the content by querying the controller. Controller *'flooding'* is the major drawback with this approach, as it is seen as the only and the central resolution entity in this solution. These solutions still carry address information of the destination domains or the services unlike general ICN approach.

2.5.7 CoLoR based controller

A controller based on *'Couple service Location and inter-domain Routing(CoLoR)'* architecture is introduced in [40]. CoLoR, as the name says, is an SDN controller that separates forwarding operations from routing operations of the network elements [84]. Like the approach suggested by Vahlenkamp et al., and Ravindran et al., the ICN network is seen as a separate domain, that provides the ICN service and, it is identified by a domain identifier which is stored in the controller. This controller is almost an extended one compared to OpenFlow, to enhance caching in the network. A controller similar to DNS is illustrated in [85], that could perform *'name-to-address'* resolution. Until the packet reaches the controller, it is handled in ICN paradigm. After they reach the controller and get resolved into the source addresses, controller informs the source about the user address, who requested the content. The source, then forwards the data directly to the user which is not the methodology suggested by ICN. Additionally, it increases the controller traffic by routing all the packets to it for resolution.

2.5.8 Cache-based approaches

Bacher et al., put forward a central SDN controller to improve *scalability* in ICN networks, that involve *live and multimedia* content dissemination. He suggested the controller to have a global knowledge about the network and to proactively cache content in the network elements that are connected to it. In this way, popular contents will be pushed down in the network near the users so that the reliability will be increased and network traffic will be reduced during popular content broadcasts. This work suggests a control plane similar to an OpenFlow control plane [86]. [80] suggests caching in the controller instead of OpenFlow switch, which may result in controller flooding and inefficient in-network caching. [82] extends POF protocol to enhance the caching capability of the switches, but remains agnostic to the implementation architecture of the switches. [87] proposes to use data centres as cache storage, instead of relying on switch's inbuilt memory. The main motto of this solution is to use the existing legacy network elements and to overcome the low storage memory issue of them, which might degrade the performance due to low line-speed processing. Some of the ICN implementations suggest the switches to cache whatever data that goes through them. Taking the hardware switch storage capacity into considerations many implementations have exercised a strategy layer that decides on what data should be cached in the switches. This decision would be carried out by running a cache management algorithm in the control plane. Wang et al., suggested one such solution by making the controller to decide on the popularity of the content using a *'linear network coding'* algorithm and additionally involving cooperative switches that communicate among them to share caching details [71] .

2.5.9 CRoS

In a completely different way from all those explained earlier, which use the existing non-ICN communication packets to communicate between switch and controller, Torres et al., introduces a controller scheme called *'Controller-based Routing Scheme'*(CRoS), that provides a control layer, that runs on top of NDN architecture and, the communication between switch and controller employs interest and data packets [88]. The controller makes routing and forwarding decisions based on the signalling information incorporated in the packets, which may eventually result in huge packets being carried in the network and, may result in network congestion. In addition, a lot of encoding and decoding efforts should be made to parse the signalling information.

2.5.10 Proxy

There are a number of past research works which relied on *'proxies'* in addition to the switches, in order to overcome the problem of switches not being able to do *'deep packet inspection'* [89] to read content names or other ICN related information in the packet. A recent work, [90] is a notable one, in which proxies with *'distributed hash table'*(DHT) are proposed, that store details about the nearest caching machines that have stored content in them. The distributed hash table is proposed to improve the scalability of the ICN network to store a large number of content names. When the switch receives a packet, it forwards it to the proxy, which extracts the content name from the packet and checks against the distributed hash table. If it identifies an entry for the nearest cache, that has stored the requested content, the packet will be forwarded to the cache to retrieve the content. If not, the packet will be forwarded to any nearest cache and in turn, the cache will forward it to the nearest server that may serve the content. In this way, the interest travels through the network towards the content producer. A proxy may get overloaded or flooded when the switch has to send all the packets to it for resolution and the solution is agnostic to the algorithms to be followed during such situations. Like DHT, which resolves the scalability issues of forwarding table entries at the infrastructure level, [36] indicated the need for a *'distributed controller'* as well, to incorporate increasing number of content names that have to be stored in routing databases.

2.5.11 Summary

This section presented various solutions in literature, to solve the problem of supporting ICN functionalities in SDN framework. Limiting the scope to one of the most successful SDN realisations, OpenFlow, this section studied the workarounds and overlay approaches applied over OpenFlow to make it support ICN functionalities without changing the underlying specification. The solutions include,

1. adding a specific tag to the packets based on the requested content name
2. substituting one of the IP header fields with ICN information
3. providing wrappers and plugins around switch and controller to support ICN packets
4. placing proxies and additional sub-nets to handle ICN packets, which the switches cannot handle
5. providing IP-based addresses to ICN domains
6. depending on costly infrastructure like data centres

The major claim of all these approaches is to realise ICN implementation in short time period using the legacy network components and by engaging workarounds for the OpenFlow limitations. Many of these approaches are successful in providing this short term ICN realisation but they result in a lot of drawbacks, when the question of standardisation and future internet architecture support put in front of them. The main drawbacks of these solutions include,

1. additional efforts to create ICN tags
2. costly encoding and decoding operations on tags
3. IP field semantic changes due to substitution with ICN information
4. processing delays due to additional modules and plugins

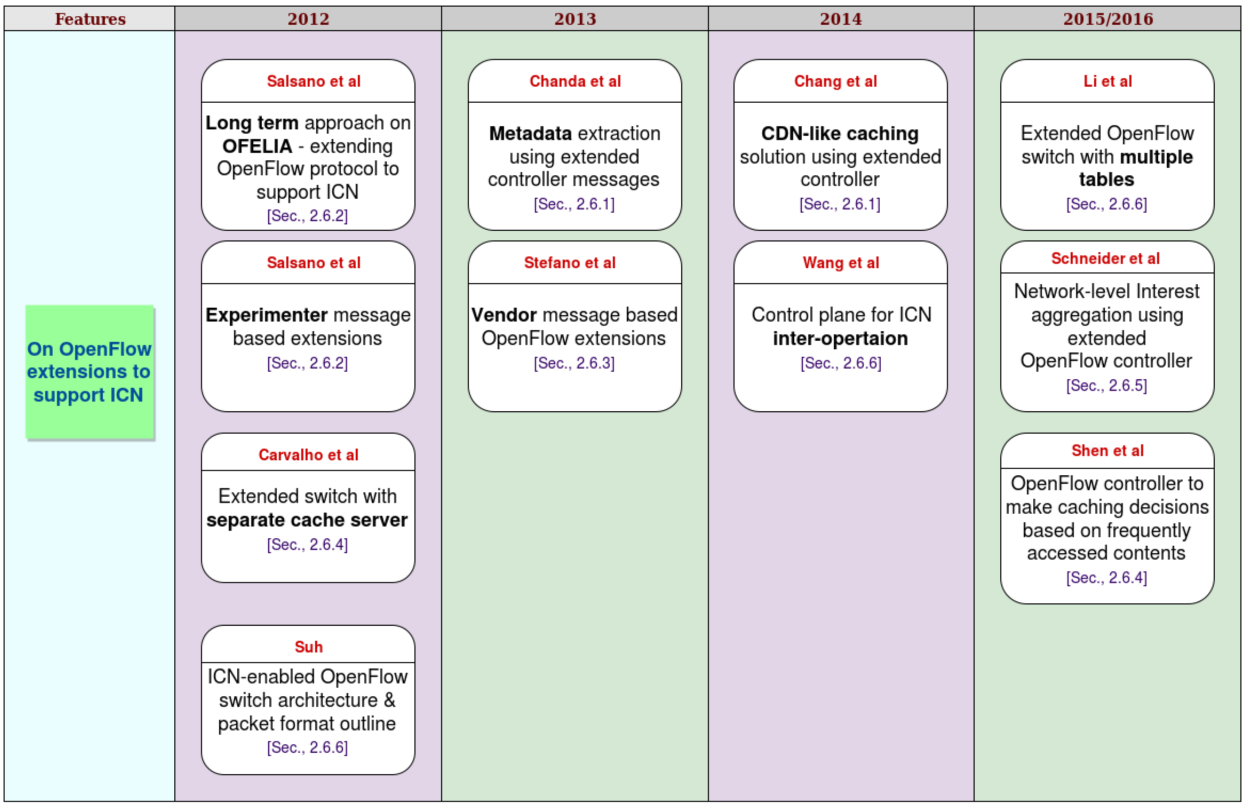
5. costly requirements to place extra machines like proxies in the network
6. still being based on IP-based addresses

OpenFlow is an evolving protocol and it is designed to operate on IP-based packets. It is true that changing or extending the OpenFlow protocol to provide support for non-IP based packets like ICN, is a complex process but, if there are no enough efforts now to modify the OpenFlow protocol to provide ICN support, it would be more difficult and would require more efforts to change it in future, when the internet architecture changes to ICN instead of IP. Moreover, these workarounds and hacks have to be revised for every new specification of OpenFlow which is again a process that requires a lot of efforts. None of the above approaches tried to provide a clean state solution for the limitations of OpenFlow in order to handle ICN flows.

Agreeing to many of these drawbacks, there are a number of studies in the literature, that tried to move OpenFlow towards a clean state realisation of ICN, by incorporating some modifications to the underlying protocol. These approaches will be analysed in the next section.

2.6 On extension based approaches

The previous section discussed the non-extension and overlay approaches to enable legacy OpenFlow implementations to support ICN related functionalities. This section covers the experiments, that tried to provide clean state ICN implementation in SDN framework, by extending OpenFlow elements; switch, controller and the OpenFlow protocol itself. Figure 2.4 shows a summary of these approaches in year based timeline along with section numbers.



Features	2012	2013	2014	2015/2016
On OpenFlow extensions to support ICN	**Salsano et al** — **Long term** approach on **OFELIA** - extending OpenFlow protocol to support ICN [Sec., 2.6.2]	**Chanda et al** — **Metadata** extraction using extended controller messages [Sec., 2.6.1]	**Chang et al** — **CDN-like caching** solution using extended controller [Sec., 2.6.1]	**Li et al** — Extended OpenFlow switch with **multiple tables** [Sec., 2.6.6]
	Salsano et al — **Experimenter** message based extensions [Sec., 2.6.2]	**Stefano et al** — **Vendor** message based OpenFlow extensions [Sec., 2.6.3]	**Wang et al** — Control plane for ICN **inter-opertaion** [Sec., 2.6.6]	**Schneider et al** — Network-level Interest aggregation using extended OpenFlow controller [Sec., 2.6.5]
	Carvalho et al — Extended switch with **separate cache server** [Sec., 2.6.4]			**Shen et al** — OpenFlow controller to make caching decisions based on frequently accessed contents [Sec., 2.6.4]
	Suh — ICN-enabled OpenFlow switch architecture & packet format outline [Sec., 2.6.6]			

Figure 2.4: Extension based approaches on OpenFlow and ICN integration - Timeline map from 2012 to 2016

2.6.1 Metadata extraction

In 2013, Chanda et al., proposed improving *'traffic engineering'* of the networks, using content names, instead of locations, by extending the OpenFlow controller that can act upon metadata extracted by additional proxies in the network [14] [91]. The switches are directed by the controller to send all the packets to the nearest proxy that is equipped to extract *'metadata'* from the packets. The metadata contains information about the content name of the data to be retrieved. The proxies communicate with the OpenFlow controller using the extended messages, in order to extract and send the metadata. Proxy, being the single point of failure and, the proxy flooding with the packets from the switches for metadata processing are the main drawbacks of this approach. Additionally, this approach operates on HTTP requests, instead of operating on ICN interest and data packets. Similar to Chanda et al., [92] proposed to extend the OpenFlow controller with new actions to invoke cache storage and retrieval. This solution heavily depends on the server locations, as the controller tries to map the name to IP address of the destination, by parsing the HTTP request. Also, the author suggests that if the controller has no information on a cache server that can satisfy the request, the controller has to forward the request directly to the destination server by reading the server name from the URL. This solution is more or less a 'Content Delivery Network' than an ICN solution.

2.6.2 Long term OFELIA approach

A long-term alternative for the short term approach proposed by [37] is introduced for the discussion, that suggested extending OpenFlow protocol and packet matching criteria to accommodate ICN methods, packets and procedures in [93]. This suggestion considered the *'CONET'* architecture for ICN and compared the *'Naming Resolution System'(NRS)* of CONET with the OpenFlow controller and pinpointed that the OpenFlow controller could be extended to act like CONET's NRS and support both IP and ICN packets in the network. [12] outlined the weakness of the OpenFlow switches to match upon arbitrary length field and introduced a fixed length tag in the packet representing the data to be fetched. OpenFlow is extended to provide ICN functionalities by exploiting the *'experimenter'* messages of OpenFlow, that operate on 'JSON' syntax [75]. OpenFlow being a binary protocol, all the messages should be represented in binary format. This work did not provide the binary equivalents of those JSON messages. These extensions based on experimenter option and JSON messages are presented in [75], that used the IP addresses to identify next hops and servers instead of the faces. This is against the basic requirements of ICN. This solution is being experimented over OFELIA testbed. This approach assumed the existence of ICN enabled switches and, remained agnostic to their implementation and architecture [69]. This approach opened the gate for the discussion on extending OpenFlow interface for supporting ICN, but the feasibility of the implementation of this approach is not discussed as part of this work.

2.6.3 Vendor messages

A less complex realisation of the above long term approach is being experimented in OFELIA, by exploiting the *'vendor message'* option in OpenFlow [94] The vendor messages are introduced in OpenFlow to carry arbitrary information. Each vendor can use this option to send their own arbitrary information between OpenFlow controller and the switch. This approach will not be helpful to realise a generic solution that can be implemented by any vendor. We feel that, using the vendor messages for each vendor to realise ICN related operations will result in interoperability issues, while integrating products from different vendors. OpenFlow is a vendor-neutral SDN standard [95]. The extensions that are proposed over it, should preserve this property of OpenFlow.

2.6.4 Cache improvements

One direction of supporting ICN in SDN by extending OpenFlow switches is introduced in [96] with a different controller and a separate cache server, to hold the cache details of the underlying network. The controller communicates with the cache server and makes routing decisions. Even though the approach suggests extending the switches to exercise ICN functionalities, it remains agnostic to the controller design. Using a controller other than OpenFlow controller with OpenFlow switches and a separate caching entity will overload the switches with additional operations to communicate with both the entities in different ways. A solution to use CCN switches, along with the legacy IP-based switches in a campus to reduce the traffic in the campus network gateway that connects to the Internet is proposed in [72]. An OpenFlow controller is claimed to be used to make caching decisions based on the frequently accessed contents. The study provides a number of caching algorithms to be exercised by the controller but remains agnostic to the implementation approach behind the controller and how the controller communicates with the CCN switches.

2.6.5 Network-level interest aggregation

ICN-enabled switch's PIT is the key player in enabling node-level interest aggregation. In a capsule view, when a request is waiting for a content in an ICN node, further requests which the node receives will not be forwarded further. Instead, those incoming requests will be stored in the PIT by making a list of waiting-faces. Researchers have proposed a further improvement in this by experimenting a network level interest aggregation. This process exploits an OpenFlow controller to decide upon few things in addition to forwarding decisions [97]. They are:

1. which interests to be stored in PIT
2. which is the best content store to store a particular content
3. which is the best path to send a content packet based on the factors; topology, cost metrics, link delay and bandwidth availability
4. how to utilise the network resources effectively

2.6.6 A view on the switch

When none of the above studies presented the ICN packet structures and switch architecture, in one of the 'NDN hands-on workshops', [13] outlined the logical structure of ICN-enabled OpenFlow switch and, ICN-based interest and data packets with the hashed content URL. This work mentions that, overlaying ICN over IP may not provide full advantage of ICN benefits. It adds that, new naming schemes and OpenFlow extensions to realise in-network caching and, new actions, may result in clean state implementation of ICN by providing an alternative to IP. A similar ICN-enabled OpenFlow Switch architecture is realised in [98], utilising multiple tables to support content storage, pending interest lists and forwarding information. The packet structure suggested in this work is same as that of the structures suggested by [13]. OpenFlow protocol extensions on messages and actions, other than caching messages are not presented by this work. This work also remains agnostic to packet matching, controller responsibilities, feasibility and implementation details of the approach. Extending the OpenFlow to allow interoperation between multiple ICN implementations is suggested in [99], where the switches assign a tag to the packets based on the ICN network from where they receive the packets. These packets will be sent to the controller and the controller makes the forwarding decisions based on packet tags and decides to which ICN domain the packet has to be sent.

2.6.7 Summary

This section analysed the existing suggestions in literature for providing a clean state ICN implementation over OpenFlow SDN framework. Each approach extended or modified the OpenFlow elements to a certain extent to provide ICN functionalities in the network. The solutions include,

1. long term experiment in OFELIA using OpenFlow experimenter messages
2. traffic engineering enhancements using proxy-based metadata extraction through extended controller messages

3. cache improvement using extended actions
4. controller directed ICN inter-operation
5. network level interest aggregation

The obvious advantage of these approaches over the non-extension approaches outlined in the previous section is that, these approaches initiated the healthy discussion of modifying the underlying OpenFlow protocol towards future internet architecture support, instead of living up with the OpenFlow limitations and using workarounds. On the other side, when we compare the amount of work being carried out in creating those workarounds and the amount of work being carried out in providing clean state ICN over OpenFlow, the latter case is minimal. Further in those minimal experiments, the scope of extension is small. Some gaps in those experiments are:

1. some experiments assumed for ICN-enabled switches and tried extending the control plane alone
2. the other group of experiments extended the OpenFlow switches but used a different controller other than the OpenFlow controller with additional communication channels
3. Few of the above solutions have used experimenter and vendor messages to experiment extensions, that might be difficult to standardise for all the OpenFlow vendors in the market
4. in documentation level, many of these works remain agnostic to implementation details
5. few studies remain agnostic to design and feasibility of the proposed architecture
6. In an abstracted level, many of these works remain conceptual without providing the feasibility results.

This dissertation aims to address many of these drawbacks. The problem formulation and the proposed solutions are captured in the next chapter.

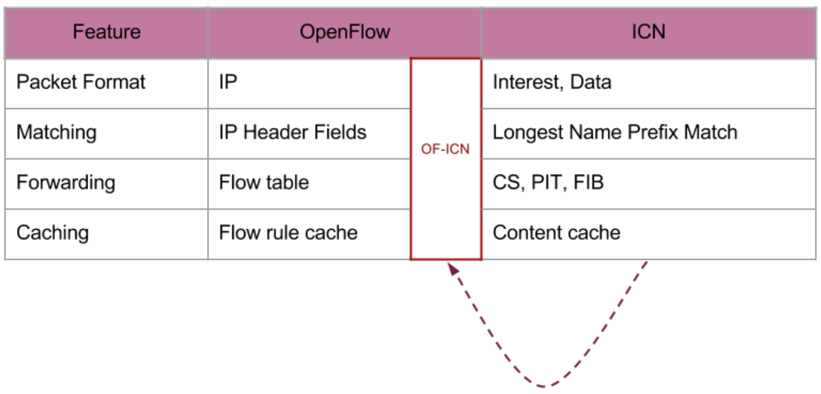
Chapter 3

Problem Formulation

The previous chapter described the literature that contributed towards porting ICN functionalities over OpenFlow and summarised the limitations and drawbacks of the same. Clearly, there is a need for a better understanding to visualise how a clean state control plane can be created for ICN, based on existing standard such as OpenFlow. This chapter presents the primary approach of this dissertation in order to show how such a control plane can be created, by considering all the necessary OpenFlow elements. The chapter starts with giving an abstracted view of the approach that will be provided to fill up the architectural differences between the two technologies under consideration. The next section talks about whereby this approach is different from many of the existing approaches, followed by the primary targets of this dissertation in order to realise the proposed approach. Finally, the challenges that are expected out of the proposed solution are listed

3.1 Abstracted view of the solution

The overall objective of this study is to enable OpenFlow to provide a control plane for the underlying ICN-enabled OpenFlow infrastructure. This study tries to realise it in a clean state way, instead of overlay approaches as listed in the previous chapter. In order to achieve that, the key solution is to fill the architectural gaps between the technologies. Figure 3.1 shows the main architectural differences between ICN and OpenFlow and it depicts how our approach 'OF-ICN' fills those differences. In a highly abstracted way, the solution is to take the features of ICN and include them within the OpenFlow architecture. By means of this approach, both ICN and OpenFlow capabilities can be achieved.



Feature	OpenFlow	ICN
Packet Format	IP	Interest, Data
Matching	IP Header Fields	Longest Name Prefix Match
Forwarding	Flow table	CS, PIT, FIB
Caching	Flow rule cache	Content cache

Figure 3.1: OF-ICN : Abstracted view of the proposed solution

3.2 Difference from existing approaches

As discussed in the previous chapter, most of the integration approaches in the industry are not providing a clean state implementation of an ICN-enabled OpenFlow control plane. Most of them are overlay approaches with extra wrappers and plugins that involve a lot of processing and cannot get the benefits of both the technologies together. A number of works tried building an ICN methodology outside the OpenFlow and yet tried connecting them through a different channel other than the normal OpenFlow channel. We believe that these approaches will not be a move towards the final aim of standardising OpenFlow to support ICN. Thus, our approach differs in a way that it builds ICN within OpenFlow by carrying out some modifications or extensions to it, in order to make it progress towards standardisation. This difference in the approach is shown in Figure 3.2.

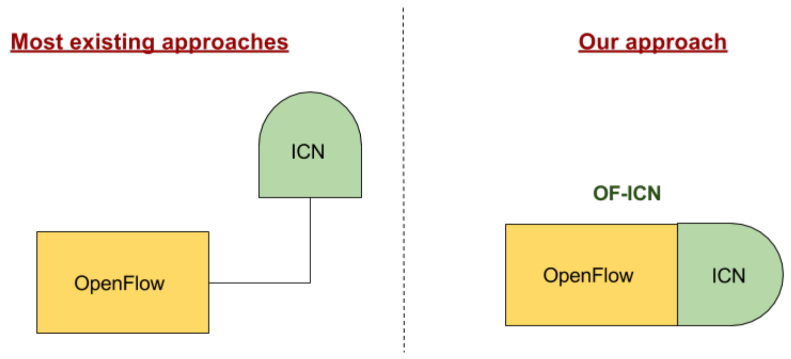


Figure 3.2: Difference between existing approaches and OF-ICN

3.3 Targets

This section lists the targets for this study to experiment with the proposed approach. This section is divided into two types of goals; design goals and implementation goals.

The design-level goals of the dissertation are:

- Use ICN-based packets which labeled information instead of addressing an end host
- Introduce faces abstraction in OpenFlow to receive and send ICN packets
- Enable OpenFlow switch to match a packet with its flow-based forwarding table using the content name in the packet
- Enable OpenFlow switch with multiple tables to support basic ICN-based data structures that are used to do packet forwarding operation
- Enable OpenFlow switch to exploit its cache to store content
- Enable OpenFlow controller to store name-based routing information and to store content in its cache
- Introduce new actions in OpenFlow switches to act upon ICN-based data structures
- Introduce new messages in OpenFlow protocol library for the switch and the controller to communicate ICN-related information

- Introduce new events and handlers in OpenFlow controller to handle ICN-related messages from switch

The implementation-level goals of the dissertation are:

- Provide a software-based experimentation code with the proposed solution
- Analyse and consider the existing extensions on OpenFlow while implementing the new extensions
- Keep the OpenFlow standards while creating new messages, actions and events
- Provide modularized code components to plugin into testing environment at any time

The following section lists some of the challenges that we foresee in the proposed approach and a number of possible solutions that can be applied to overcome these challenges. The later chapters will discuss how well the proposed approach handled these challenges as part of the experimentation.

3.4 Challenges

OpenFlow and ICN put forward some challenges for them to be integrated in order to support ICN flows. These challenges are listed below:

- **OpenFlow for non-IP**

  OpenFlow is designed to operate over IP packets and match against the IP and transport protocol header fields. So far, OpenFlow has been extended to provide support for VLAN tags, MPLS tags and to support circuit switching, which is again realised as overlays over IP [70]. When OpenFlow was announced, it was claimed to support non-IP packets as well, but there are only very few experiments over it which almost revolve around IP-based packets. We try to change the OpenFlow switch behaviour so that it can recognise a non-IP packet, by constructing the packets as raw Ethernet packets and making the switch to deal with them

- **Binary protocol**

  OpenFlow switch and the controller communicate through a secure channel and OpenFlow is a binary protocol where all the messages are packed into binary code and unpacked at the receiving end, into the original format [58]. This is to ensure the additional security of the messages on the wire. The events, which the switch sends to the controller and the actions which the controller sends as messages to the switch are all get converted into binary codes and converted back into normal format at the receiving end. So, at the implementation level, the new messages and actions should be packed properly so that the receiving end will be able to parse them and invoke modules based on them. This is a difficult process as a single bit change in the message packing may send a wrong message to the controller and the packet may not be processed on time. This will result in more packets getting dropped and can affect the quality of service of the underlying network. Our study analyses the packing and unpacking methodology being used in OpenFlow for different types of messages and implement those methodologies over new messages and actions being created as part of the experimentation

- **Variable name size**

  This is a challenge put forward by both ICN and OpenFlow. Unlike IP prefixes, which are of fixed size, ICN names are variable in size and there is no proper standardisation on the ICN naming schemes yet. In addition, OpenFlow is not equipped yet to match a variable-sized header field with its flow table. It can only match with fixed sized header fields as with IP packets. A number of solutions are there to support ICN names in OpenFlow by hashing or tagging the name into an IP field, as mentioned in the previous chapter, but still, there is no solution to make OpenFlow support an arbitrary name field. A proper solution for this issue can only be provided when the naming methodology is standardised for ICN. As part of this experimentation, we are making the switch to traverse the content names in the packets assuming that the switches are equipped to do deep packet inspection.

3.5 Summary

This dissertation addresses the problem of providing a clean state integration of OpenFlow and ICN so that an OpenFlow-based control plane can be created for an ICN network. This dissertation analyses the ways to create such a control plane for ICN, by studying the existing literature works on integrating OpenFlow and ICN and, by plotting the architectural differences between OpenFlow and ICN. It proposes a solution called 'OF-ICN', wherein the ICN functionalities are ported over in OpenFlow architecture, by extending the necessary OpenFlow elements and through which, it derives an ICN-enabled OpenFlow control plane for the underlying ICN-enabled OpenFlow switches.

Chapter 4

Design

This chapter presents the design choices that are proposed for this dissertation. The first section describes the functional architecture, by depicting the main functional topology, that we use throughout the experimentation. The second section portrays the architectural design of the modified OpenFlow switch. It also details the proposed messages and actions that are added to the switch functionality. The third section portrays the design of the modified controller along with the information on messages that are added or extended to the existing controller functionalities. The last section talks about the packet design that we use for the study and the chapter concludes with a summary.

4.1 Functional architecture

Figure 4.1 shows the network topology that we use for experimenting the proposed OF-ICN approach. As shown in Figure 4.1, the topology has two hosts: *'Host 1'* and *'Host 2'*. Either of the hosts can be a consumer or producer of a content. For most of our experiment evaluations, Host 1 will be the consumer and Host 2 will be the producer. There are two OpenFlow switches, that are part of our design. These switches are modified to enable ICN capabilities as per our approach. The switches are named as *'ICN Switch 1'* and *'ICN Switch 2'* and referred as 'Switch 1' and 'Switch 2' in further chapters for light reading purpose. As shown in Figure 4.1, each modified OpenFlow switch has the following three basic ICN-based elements to support ICN flows: *Content Store(CS), Pending Interest Table(PIT) and Forwarding Information Base(FIB)*. Host 1 is connected to Switch 1 and Host 2 is connected to Switch 2. Both the switches are connected to each other as well. The switches are also connected to the modified OpenFlow controller as shown in Figure 4.1. The modified OpenFlow controller contains the following elements in order to make forwarding and caching decisions for ICN-enabled OpenFlow switches: *cache and ICN-based routing database*. The design of the switch and controller components are explained in following sections.

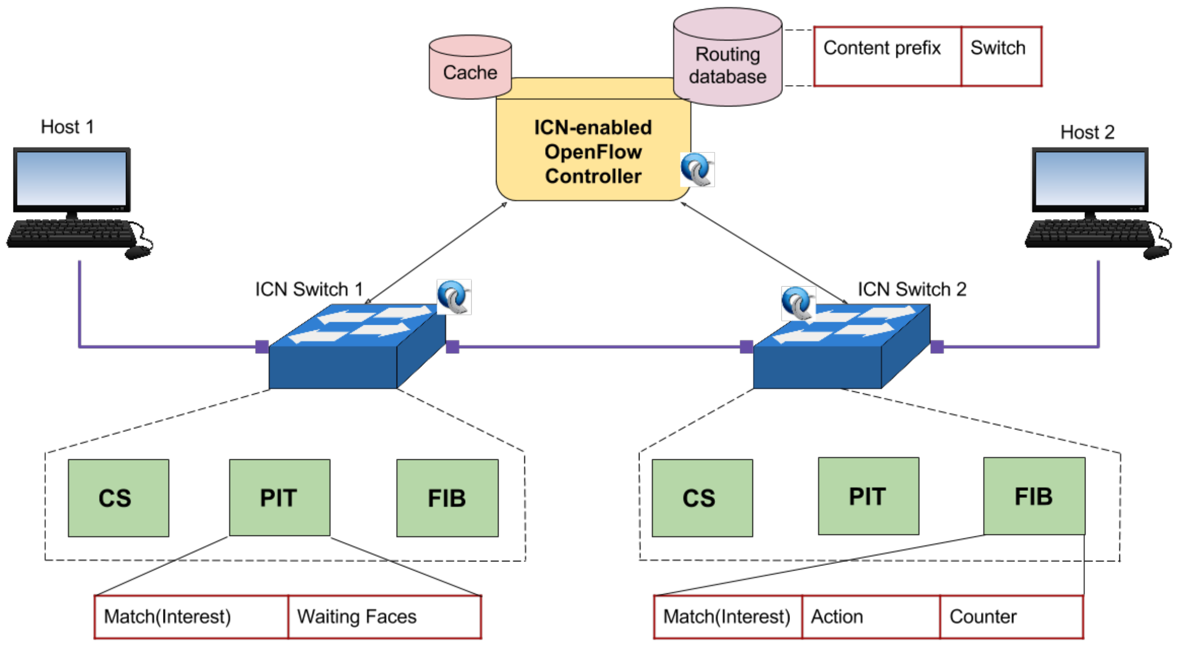


Figure 4.1: Functional architecture : OF-ICN design components

4.2 Switch design

This section presents the architectural and functional details of the new switch elements that are introduced as part of our solution, followed by the details on the messages and actions, that are introduced to the modified switch.

4.2.1 Forwarding Information Base

Every ICN-enabled OpenFlow switch(OF-ICN switch) has a *Forwarding Information Base*(FIB) table which helps in making forwarding decisions at the switch level. Each entry in FIB is a combination of OpenFlow match, OpenFlow action and counter fields. The *'match'* field contains content names to match against the incoming packets. If the content name in the interest packet matches an entry in the table, then the corresponding list of OpenFlow actions in the *'action'* field will be executed on that packet. An action can instruct the switch to send the packet out in a particular face. 'OF-ICN' switch supports a number of ICN-related actions which will be listed in later sections. The *'counter'* field of an FIB entry will be incremented upon every packet that matches with that entry. Figure 4.2 shows the FIB table structure.

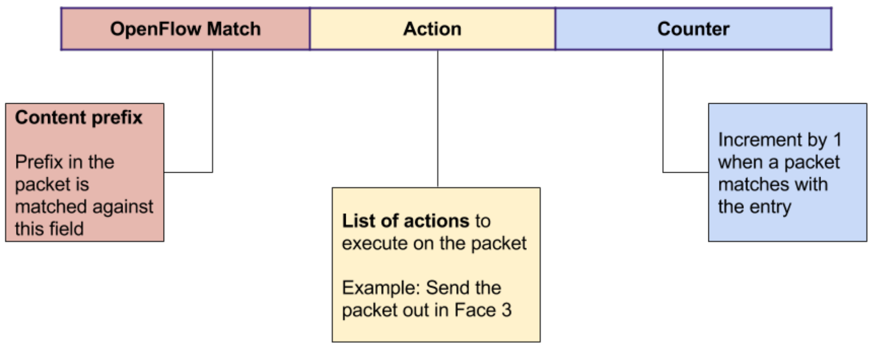

OpenFlow Match
Action
Counter
Content prefix
Prefix in the packet is matched against this field
List of actions to execute on the packet
Example: Send the packet out in Face 3
Increment by 1 when a packet matches with the entry

Figure 4.2: FIB structure

4.2.2 Pending Interest Table

Pending Interest Table(PIT) is the one that helps the switch to save *'breadcrumb'* for the interest packet that is sent upstream towards the content. As the name suggests, PIT stores the pending interests which have to be satisfied with contents. Through this table, the contents consume the interests. PIT consists of two fields: *'OpenFlow match'* and *'waiting faces'*. Like FIB, the match field contains content names to match against the content packets. If a content packet matches with an entry, it will be sent in the faces that are waiting for it. The switch adds the incoming face to the waiting list, when an interest packet is forwarded towards an upstream face. Figure 4.3 shows PIT structure.

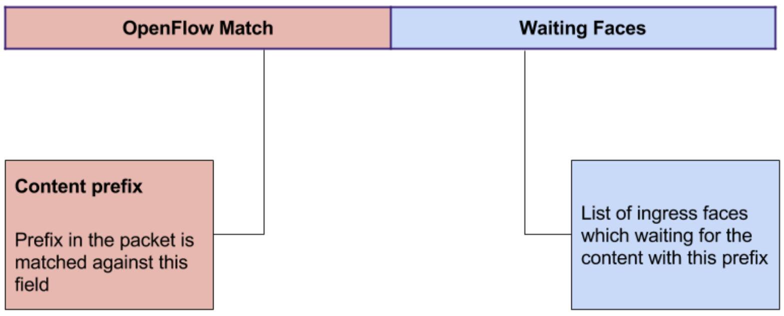

OpenFlow Match
Waiting Faces
Content prefix
Prefix in the packet is matched against this field
List of ingress faces which waiting for the content with this prefix

Figure 4.3: PIT structure

4.2.3 Content Store

Content Store(CS) is a logical view of switch's *'internal cache'* to store the contents. Each entry in the content store contains an 'OpenFlow match', that denotes the content name that can match with the packet and, *'data'*, that is the content that can satisfy the interest. If an incoming interest packet matches an entry in the content store, the corresponding data will be sent back towards the incoming face. Content store is populated by a number of ways: 1) When a switch receives a data packet corresponding to an interest packet, it can store that content in the cache 2) The switch can store the contents announced by the hosts that are connected to it 3) The controller can push the content down to the switch for it to satisfy any upcoming interests for that content. Figure 4.4 shows the 'Content Store' table structure.

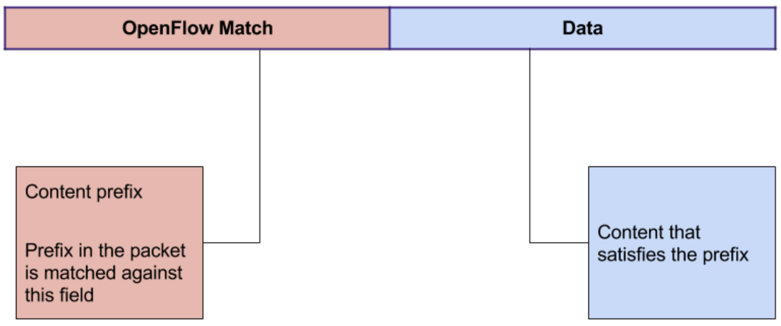

OpenFlow Match
Data
Content prefix
Prefix in the packet is matched against this field
Content that satisfies the prefix

Figure 4.4: Content Store structure

4.2.4 Face

The *'face'* is an abstraction of an interface through which the switch can receive or send packets in the network. A face can be connected to an internal application, an Ethernet wire or a device. Every face in the switch is identified by a fixed length identifier. The controller has the overall view of the faces that a switch has and, uses it to take forwarding decisions. In Figure 4.1, Switch 1 is connected to Host 1 through F1(Face 1) and it is connected to Switch 2 through F2(Face 2)

4.2.5 Extended messages

The new messages that are introduced as per our solution and the existing OpenFlow messages, that are modified as per our requirement are listed below. These messages are triggered by the modified OpenFlow switch:

CONTENT_ANNOUNCEMENT

OF-ICN switch invokes this message to the controller in order to register the availability of a content in the network. The switch can send this message to the controller whenever it receives a *'content publication'* message from a host or the switch caches a content in its content store. The message contains the content prefix and the switch's identifier.

The controller extracts this information from the message and stores them in its routing database. Thus, this message helps the controller to study the switch that is responsible for a particular content.

CS_FULL

OF-ICN switch sends this message to the controller, whenever the content store is full. This can be realised anytime when the switch tries to add a new entry to the cache and finds that the cache is full. This message plays the role of an *'event notifier'* and helps to delegate the caching decision responsibility to the controller.

FIB_FULL

The memory capacity differs between the switches. The low-capacity switches are expected to store a limited number of flow rules in their FIB [26]. In this case, the controller has to take the major responsibility of deciding on the flow rules that have to be stored in the switches. Moreover, when the switch's FIB is full, the controller has to take some actions to manage the FIB. For example, the controller can ask the switch to clear the FIB or it can ask the switch to delete the least used entry from the FIB. In order to achieve this, the switch has to inform the controller whenever its FIB is full. This FIB_FULL message is useful for this purpose and helps the switch to notify the controller when the switch realises that its FIB table is completely filled and there are no more spaces to fill new entries.

FIB_REPLY

The switch sends this message as a reply to the FIB request message that is sent from the controller. The controller usually requests for FIB_REPLY when the switch connects it or at any time it wants to recalculate its routing information. The FIB_REPLY message carries the flow entries from switch's FIB table which includes the following fields: content name, actions list and counter.

FIB_MOD_NOTIFICATION

Whenever the switch makes some changes to the FIB, it has to notify the controller about the modification. This helps to keep the global view maintained by the controller to be in sync with the underlying datapath. This message is sent by the switch, when it adds an entry to the FIB, deletes an entry from the FIB or makes some changes to the existing flow rules based on FLOW_MOD messages from the controller. This message is also considered as an *acknowledgement* from the switch in response to the flow rule pushed by the controller to it.

FEATURES_REPLY

This is an existing OpenFlow message which is used to announce the capabilities of the switch to the controller [6]. This is modified based on our requirement and allows the switch to announce ICN-related capabilities to the controller. The ICN-related capabilities include; the number of faces supported by the switch, the number of tables supported and the caching capability of the switch. This can also be extended to include more information. For example, the switch memory capacity can also be announced as part of this message. This message is triggered as a response to the FEATURES_REQUEST message from the controller during the switch-controller handshake phase.

4.2.6 Extended actions

The new actions that are introduced as per our solution and the existing OpenFlow actions, that are modified as per our requirement are listed below. These actions are executed by the modified OpenFlow switch:

ADD_PIT

OF-ICN switch invokes this action to add an entry into its 'Pending Interest Table'(PIT), whenever it forwards an interest packet in an output face. This action takes a content prefix and a list of faces and, adds them into the PIT as a new entry. This is an internal action invoked by the OpenFlow switch and not exposed to the controller

REMOVE_PIT

When the OpenFlow switch receives a data packet, it retrieves the faces from PIT, that are waiting for that content. The data packet will be sent in these waiting faces. Once the data packet is sent out, the switch invokes the REMOVE_PIT action to remove the corresponding PIT entry, as the interest is already satisfied by the content. This action takes the content prefix as a parameter and deletes the corresponding entry from PIT

OUTPUTFACE

The switch executes this action to send interest packet or data packet out in a face. This action takes a *'face id'* as the parameter. OpenFlow controller adds this action into an FIB_MOD message and sends it to the switch. This action will be stored in the action field in the corresponding FIB entry. When an interest packet matches the corresponding entry in FIB, the switch executes this action, retrieves the face and send the packet out in that face. During the transmission of a data packet, the switch will pass the 'waiting face id', that is retrieved from the PIT to this action. The action will send the data packet in that face back to the requester.

CLEAR_CS

This action is invoked, when the switch receives 'CLEAR_CS' message from the controller. The switch receives the instruction to clear out the content store, as a response to the notification of the switch to the controller when the content store is full. This can also be a direct instruction from the controller, if the controller expects to push some popular multimedia content to the switches and it wants to clear the caches from the switches. This action deletes all the entries from the switch's content store.

DELETE_LEAST_FIB

This switch action is triggered when the controller sends an instruction to delete the FIB entry that is used the least among all the FIB entries. The switch receives such an instruction as a response to the notification sent by the switch to the controller when its FIB is full. The controller can also send an instruction to make some space in the switch's FIB in order to push a new flow modification rule. In both the cases, the switch queries for the least value in the counter field in FIB table and deletes the corresponding entry

4.3 Controller design

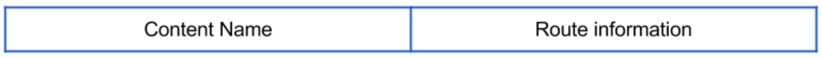
Controller is the entity, that handles forwarding decisions for the underlying datapath infrastructure and there are many SDN-based controllers available in the market [60]. OpenFlow, being one of the leading SDN innovations, has its own controller to control underlying OpenFlow switches [3]. The OF-ICN controller is equipped with a *'Routing database'*, which can be the superset of the forwarding tables of all the switches that are connected to it. The controller uses this routing database to decide on where to send a packet, which the switch has sent to it. Based on this decision, the OpenFlow controller pushes flow rules to the switches that are in the route path and, sends the packet back to the corresponding switch that queried the controller. Then the switches in the path, can forward that packet based on the rules sent to them by the controller.

We are using the OpenFlow controller, but extend it to perform ICN related functionalities over ICN packets. The following sections describe the new messages and actions that are supported by our OF-ICN controller.

4.3.1 Routing database

'Routing database' is same as that of a routing table in the legacy OpenFlow controller and it stores routing information based on the content names. This table is used to store

reachability information for every content name in the network that the controller knows about. For example, it can store the identifier of the switch, that has cached the content or, that is the nearest one to the content producer. This information is expected to be populated by any *name-based routing protocol* (for example, NLSR [50]) that runs on the network. This database will also be updated based on the content announcements from the switches connected to the controller. The controller queries this database whenever a switch sends an interest packet to it, based on the content name available in the packet. The controller will push the flow rules based on the routing information present in the routing database for that content name. In order to achieve this process, the necessary fields that are considered as part of our routing database design are : *'content name'* and *'route'* associated with the content name. The route can be the switch identifier which is connected to the controller, that announced the content to it or it can be a number of intermediate switches that form a path to reach the content. The routing database structure is shown in Figure 4.5.



Content Name	Route information

Figure 4.5: Structure of controller routing database

4.3.2 Controller cache

One of the main objectives of ICN is to enable *'in-network caching'* in the network, to improve the availability of the content near the end-users. Even though the switches are expected to store some content in their cache to provide better availability in the network, the storage differs between the switches and it depends on the memory capacity of the switches [25]. We can not expect the switches to store a lot of content, that keeps increasing day by day. We propose the OF-ICN controller to exploit the cache in the machine where it is running, to store the content published by the switches and hosts.

This is to increase the content availability of many switches which are connected to the controller, as they can retrieve the required information from the controller itself, instead of reaching out to the content producer. When a controller receives an interest packet from a switch, it checks its own cache for the availability of the content. If the content for that request is available in its cache, the controller can send the data directly to the switch to satisfy the content requester as soon as possible. This can significantly reduce the *Round Trip Time*(RTT), otherwise required if the switch has to reach out to any other in-network cache or the content producer. This cache can also be used to push content to the underlying switches in a *'proactive'* way when the controller suspects an upcoming huge need for a particular content (for example, upcoming popular show broadcast). The controller cache also has a field called *'priority'*, that denotes the popularity of a content. If the controller receives a popular multimedia content, it will store that content in its cache with a possible highest priority. An algorithm that runs in the controller frequently monitors the priority of the contents in the cache and proactively pushes the highest priority content to the underlying switches, to reduce transmission delay in the network. The structure of controller cache is shown in Figure 4.6.

Content Name	Data	Priority

Figure 4.6: Structure of controller cache

4.3.3 Extended messages

The new controller-triggered messages, that are introduced as per our solution and the existing controller-triggered OpenFlow messages which are modified as per our requirement are listed below:

PACKET_IN

In OpenFlow, this is the message which the switch sends to the controller if it has no matching entry for a packet in its flow table [6]. This message is extended to send the *content name* to the controller, in addition to other protocol fields. The controller extracts the content name from this message and, uses it to query its cache and routing database, for any information on that content name.

ADD_CS_ENTRY

This message allows OF-ICN controller to disseminate content to the switches in a proactive way. As explained earlier, when the controller is expecting a huge demand for a content or it has identified the next content in the sequence for a previously requested interest, it pushes the content down to the switches for them to store in their caches. This will help to improve the availability of the content near the users during the high content demand scenarios. The ADD_CS_ENTRY message contains the content prefix and the corresponding content. These details are sent from the controller to the switch in the binary format.

CLEAR_CS

This message helps OF-ICN controller to make the switch clear its cache. This message can be invoked as a response to CS_FULL message from a switch. It is also a part of the algorithm, that runs in the controller and makes caching decisions for the underlying switches. For example, when the controller has some popular content to push to the switches and, when the cache in a switch is filled with less popular contents, the controller can send this message. It will instruct the switch to clear its cache so that the controller can push new data to it.

DATA_FROM_CONTROLLER_CACHE

This message is utilised by the OF-ICN controller to send data back to the switch, that requested for an interest packet resolution and, the controller itself has the data for that interest in its cache. This message contains the content prefix and the corresponding data to be sent to the requested switch. When the switch receives this message it will invoke two actions. One is to send the data back in the waiting faces and the other one is to add the content to the cache. This caching decision can be made locally in the switch or by the controller

FIB_MOD

This message is an extension of OpenFlow's 'FLOW_MOD' message [7] supports pushing down fib flow rules from the controller to the switch. This message is based on the information retrieved by the controller, from the routing database, for a particular interest sent by the switch to it. The controller frames the FIB_MOD message with the content prefix and the *next-hob face* for the switch to sent out the interest packet. This message is pushed to the switch and gets added into its FIB, if there is no existing entry for that content prefix. Otherwise, it modifies the existing entry in FIB with the new rule.

FIB_REQUEST

The OpenFlow controller is expected to have a global view of the underlying datapath [22], which is a group of connected switches in our case. When a switch initializes with the controller, it informs the controller about the contents which can be reached through it. This can be compared with a *'publishing'* operation. This is achieved by informing the FIB details of the switch to the controller. Thus, when the switch connects to the controller, the controller sends an FIB_REQUEST to the switch, asking it to send the FIB details in an FIB_REPLY message. Not only during the handshake, at any time when the controller needs to get FIB details from a switch to make any controlling decisions, it can send out this message and, get the FIB details back in the reply from the switch.

DELETE_LEAST_FIB

This is a response message from the controller for the 'FIB_FULL' message from one of the connected switches. This message instructs the switch to find out the FIB entry which is used least among all the FIB entries and delete the entry. In our approach, this message is expected to be triggered only when the switch sends an 'FIB_FULL' message, but this message can also be used by the controller to clear out some FIB entries in the connected switches before it pushes FIB_MOD entries for any new high-demand content.

4.4 Packet design

All the different ICN implementations share one basic feature in common, which is to support ICN-based packets [80]. In a most general view, an end user in an ICN network sends out a request for a content in terms of an *'interest'* packet and gets back the content in terms of a *'data'* packet. The another important packet type that is used to publish or announce the content in ICN network is *'content announcement'* packet, which helps to register the content in the ICN network. Following this generic ICN approach, our solution also comes with these three basic ICN-based packets; Interest, Data and Content Announcement. Limiting the scope of this study to the basic ICN functionalities, our packets carry the necessary information to enable those functionalities in the network. As shown in Figure 4.7, an 'interest' packet contains the content name(prefix) of the content to be fetched, along with the packet headers. A 'data' packet contains the content name and the actual content(data) corresponding to that name. The 'content announcement' packet contains the content name which is being registered and the identifier of the switch which is sending the announcement message to the controller on behalf of an end host. The message packets which are communicated between the switches and the controller retain the message structure proposed by OpenFlow [6]; OpenFlow header which identifies the type of the packet, followed by the payload which contains the actual information.

Packet Type	Structure			
Interest Packet	Packet Headers	Content Name		
Data Packet	Packet Headers	Content Name	Data	
Content Announcement	Packet Headers	OpenFlow Header	Content Name	Switch Identifier
Any OpenFlow packet	Packet Headers	OpenFlow Header	Payload	

Figure 4.7: OF-ICN packet structures

4.5 Reactive processing algorithm

This section delineates the steps carried out by OF-ICN using the proposed design to fetch a content from the network. The controller sends the flow rules on a request from the switch, which is a *'reactive'* process. The steps are visually shown in Figure 4.8 and they are explained below:

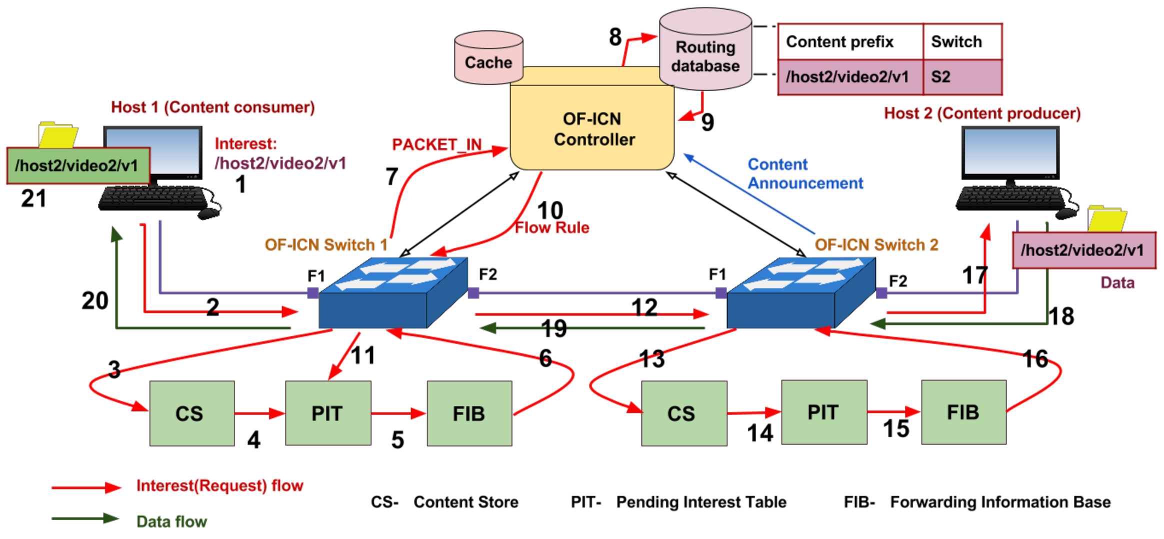


Figure 4.8: OF-ICN reactive processing sequence

- **Scenario :** The topology is depicted in Figure 4.8. 'Host 2' produces a content named '/host2/video2/v1' . 'Host 1' requests for the same content by sending a request to the switch 'Switch 1', to which it is connected to.
- **Bootstrap :** We assume that Host 2 has already published the content that is named'/host2/video2/v1', to 'Switch 2', as it is connected to Switch 2. Switch 2 has added an entry in its FIB, with the content name and the face that connects Host 2, as the next-hop route. Switch 2 has also announced this content to the controller. Thus the controller has the routing information for the content as shown in Figure 4.8.
- **Step 1 :** Host 1 creates an interest packet with the content name '/host2/video2/v1'. This interest packet can be compared to a request (say., an HTTP request). In our case, it is an ICN Ethernet packet with the content name.
- **Step 2 :** Host 1 sends the interest packet to Switch 1 as it is connected to it.
- **Step 3 :** Switch 1 extracts the content name from the packet. It checks in its 'Content Store' for a matching entry by comparing the content name with the 'match' field. In our case, the 'CS' has no entry for '/host2/video2/v1'.
- **Step 4 :** Switch 1 checks the 'Pending Interest Table' if it has any interests which are already waiting for the same content. As this is the first interest, there will not be an entry in PIT in this case.
- **Step 5 :** Switch 1 checks the 'Forwarding Information Base' if it has the forwarding route for the content name. In our case, Switch 1 will not have a matching entry in FIB.
- **Step 6 :** No match in any of the switch tables.
- **Step 7 :** Switch 1 prepares a 'PACKET_IN' message with the extracted content name and sends it to the controller.

- **Step 8 :** Controller checks for the routing information for the content name, in its routing database.
- **Step 9 :** Controller gets back the routing information. In our case, the identifier of Switch 2 will be returned for the content '/host2/video2/v1'.
- **Step 10 :** Controller pushes a 'FIB_MOD' message to Switch 1, stating that, for the interest packets named '/host2/video2/v1', Switch 2 is the route. It also sends back the interest packet to Switch 1.
- **Step 11 :** Switch 1 stores the FIB rule from the controller and adds a PIT entry for the interest, before forwarding it to Switch 2.
- **Step 12 :** Switch 1 sends the interest packet out in the face which is connected to Switch 2.
- **Step 13 :** Switch 2 receives the interest and checks in its CS for a matching content. In our case, Switch 2 has not cached the content yet.
- **Step 14 :** Switch 2 then checks in its PIT for any pending interests for the same content name. Because this is the first interest, there will not be an entry in PIT.
- **Step 15 :** Switch 2 checks in its FIB for the next-hop information.
- **Step 16 :** Switch 2 finds out that the face, that connects to Host 2 is the next-hop route for the interest with the content name '/host2/video2/v1'.
- **Step 17 :** Switch 2 sends the interest packet out in the face that connects to Host 2.
- **Step 18 :** Host 2 recognises the interest packet, as it produces the content. It creates a data packet with the content and sends back to Switch 2.

- **Step 19 :** Switch 2 sends the data packet back to Switch 1 in the face that connects to it. This is possible because Switch 2 would have stored a PIT entry with the face that connects to Switch 1 as waiting-face, before forwarding the interest to Host 2.
- **Step 20 :** Switch 1 receives the data packet. It checks its PIT and retrieves the waiting face, which is the face that connects to Host 1. It sends the data packet out in the face that connects to Host 1.
- **Step 21 :** Host 1 receives the data, which it requested for.

4.6 Summary

OF-ICN design is presented in this chapter. We port the basic ICN-based data structures into OpenFlow switch and OpenFlow controller, to enable them to process ICN packets. We introduce new messages and actions to OpenFlow protocol, to communicate ICN-related information between the switch and the controller. A functional topology is shown in Figure 4.1 with the modified OpenFlow elements:

- OF-ICN switch (ICN-enabled OpenFlow switch)
  - **Content Store** : To store ICN contents (data objects)
  - **Pending Interest Table** : To store the information about the interests, that are sent upstream in the network
  - **Forwarding Information Base** : To store next-hop forwarding details for content names
- OF-ICN controller (ICN-enabled OpenFlow controller)
  - **Routing database** : To store routing information for content names
  - **Cache** : To store contents in the controller

- End-hosts : End hosts are enabled to send out ICN interests and to receive ICN data packets

The architecture and the functionality of all the extended components are briefed in this chapter. New messages and actions that are introduced to OpenFlow as part of our design are categorized into *'switch-based'* and *'controller-based'* and, their architecture and functionality are detailed under the corresponding category. The packets we use in OF-ICN are divided into following categories: **Interest**, **Data**, **Content Announcement** and **OpenFlow message packets**. The structure of each of these packets are portrayed. In order to derive the basic functionality over the proposed design, an algorithm is presented for **'reactive processing'**, using the OF-ICN switches and controller, along with a diagram, depicting the sequence of communications takes place in the process.

Chapter 5

Implementation

This chapter describes the implementation details of this dissertation. It starts with a section that outlines the technical architecture of the experiment, followed by the sections that explain the test environment and different components involved in the implementation. This chapter also speaks about the methodology to create new messages and actions which are proposed as part of our approach, followed by the basic commands to run the code from GitHub. A summary concludes the chapter.

5.1 Technical architecture

Figure 5.1 shows the technical architecture of our implementation. The figure demonstrates the switch and controller components using a Python implementation of an OpenFlow controller, POX. This section presents the environment, technologies used and the components involved in the implementation. Figure 5.1 is divided into two sections. On the left, it is the OpenFlow switch components and on the right, it is the OpenFlow controller components.

The following section introduces the main platform POX, which is used for our experiment and about its core components.

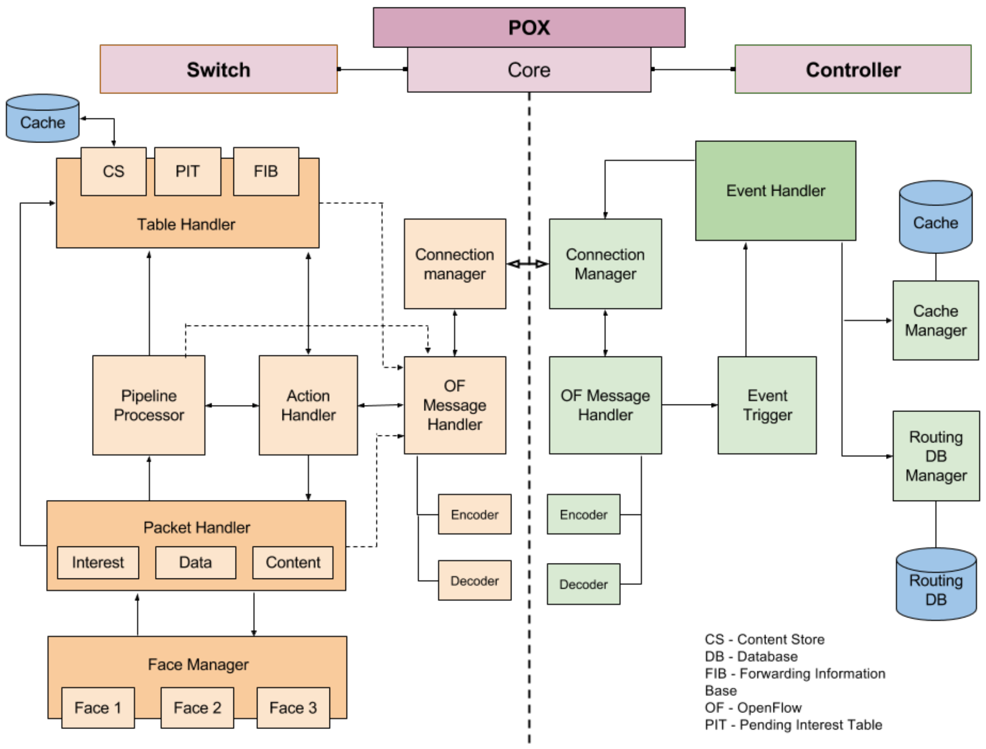


Figure 5.1: Implementation Architecture - Switch (Left side) and Controller (Right side) components

5.2 POX

POX [62] is a realisation of an OpenFlow controller, in Python language. It is a networking framework used to prototype and experiment SDN-based networking architectures and applications. It is derived from NOX, a popular OpenFlow controller implementation in C++. POX works on component-based methodology in which every module is visualised as a component that is registered into POX core, which is again a parent component. The current stable version of POX supports OpenFlow 1.0 wire protocol and nicira extensions [58], to support a number of OpenFlow 1.2 features. Even though POX initially provided only an OpenFlow API and controller implementations, it now has basic switch datapath implementations as well. Thus, we utilised POX to develop both switch and controller components. Our main motto of extending the OpenFlow interface is realised with a new set of messages and actions, by modifying the OpenFlow library provided by POX .

5.2.1 Core

As mentioned before, POX requires the modules to be registered as components into its 'core' component. When POX boots, it prepares the core components and attaches all the registered components to it. This is helpful for the components to refer or use other components by just retrieving them from the core. Multiple instances of a component can also be registered with the core object. We have created a controller component and a switch component and added them to the POX core object. Each component can have multiple modules to support its functionality. Our controller component has the following modules: Connection manager, OpenFlow message handler, event trigger, event handler, cache manager and routing database manager(Topology manager). Similarly, our OpenFlow switch component has the following modules: Connection manager, OpenFlow message handler, action handler, pipeline processor, table handler and the packet handler. These modules are illustrated in figure 5.1 and they will be explained in the following sections.

The following code snippet from our implementation shows how an instance of a switch module can be added to the POX core object

```
def icnswitch1 (dpid = "123"):
  """
  Launches an ICNSwitch
  """
  from pox.core import core
  core.register("datapaths", {})  #This adds current instance of the
                                   switch module to POX core under
                                   the name "datapaths"
```

Adding a component to POX core, also helps a component to listen for an event from the other component just by mentioning the event name instead of mentioning the component name. An event is a special kind of notification based on OpenFlow messages which will be explained in the forthcoming sections. The following code snippet from our controller module shows how the controller listens for 'ConnectionUp' event, that is triggered when a switch connects to the controller:

```
core.openflow.addListenerByName("ConnectionUp", start_switch)
#here 'start_switch' is the event handler function that has to be called
 for ConnectionUp event
```

5.3 Test environment

Keeping the POX networking platform as the base, the following components or entities are required for the demonstration of our approach:

- OpenFlow software switch

- OpenFlow controller implementation
- OpenFlow interface

In order to simplify the complexity of implementation and to provide better clarity in understanding, we exploited the fact that POX provides support for all the above components through its component-based architecture.

Installation requirements:

We have used a 64 bit single machine with Ubuntu 14.04.4 LTS(trusty) installed in it. This Ubuntu distribution is downloaded from *http://releases.ubuntu.com/14.04/*. Ensuring that the user has administrative privilege is very important as many packages, modules and processes in the setup, run with admin permissions. The packages which are required in order to run our setup are listed down along with the terminal commands to download and install them:

- ***python-setuptools*** : *sudo apt-get install python-setuptools*
- ***JDK*** : *sudo apt-get install default-jre , sudo apt-get install default-jdk*

  Ensure that either open JDK 7 or Open JDK 8 is installed by the previous commands. This can be checked using *'java -version'* in the terminal. If not, install the required JDK version by explicitly mentioning it in the command. For example, to install JDK 8 use the commands: *sudo apt-get install openjdk-8-jdk , sudo apt-get install openjdk-8-jre*
- ***Git*** : *sudo apt-get install git*

  Git is required to get the basic setup files from the respective repositories. After installing the git, configure it using the following commands: *git config –global user.name "username" , git config –global user.email "email"*
- ***PyCharm*** : *sudo apt-get install pycharm*

All our setup and development files are in Python. An IDE like 'PyCharm' is helpful to write better python code as Python is very strict about code formatting like 'indentations'. The above command would install premium version of Pycharm with 30 days trial period. The following command is used to install free version of PyCharm : *sudo apt-get install pycharm-community*

- ***Wireshark*** : *sudo apt-get install wireshark*

  Wireshark is a packet analyser used to dissect and inspect the packets transmitted through the network [100]. If POX is installed through mininet, mininet also has a provision to install Wireshark along with it. Otherwise, it can be installed using the command given above. Once the Wireshark is installed, it can be invoked using the command *'sudo Wireshark'* from the terminal

- ***tkinter*** : *sudo apt-get install python-tk*

  Tkinter [101] is a GUI tool to visualise python dictionaries and tables in different views like 'TreeView'. By default, many Python installations come along with Tkinter package. If not, it can be installed using the command given above

- ***POX*** : *git clone https://github.com/noxrepo/pox.git*

  POX can be run as a standalone networking framework as it provides all the necessary components like OpenFlow software switch, OpenFlow controller and protocol API in terms of libraries. POX controller is installed by default while installing Mininet or it can be installed using the source code from git repository using the command given above. If the above command is executed from the home directory, it will create a directory named 'pox' with all the necessary source code in them. The latest stable release of POX when this study is carried out is *'carp'*

5.3.1 Logical entities

Our setup has the following entities for the demonstration of the proposed approach:

- Two ICN-enabled OpenFlow switches: Switch 1 and Switch 2. Switch 1 and Switch 2 are connected to each other
- One ICN-enabled OpenFlow controller: Both the switches are directly connected to the controller
- Two hosts : Host 1 and Host 2 . Host 1 is connected to Switch 1. Host 2 is connected to Switch 2

5.4 Connections

We studied the basic installation steps and the logical entities that are required for the implementation, in the previous sections. This section shows how they are connected to each other in our experiment setup.

Both switches, Switch 1 and Switch 2, are connected to OpenFlow controller directly using OpenFlow interface. The switches communicate with the controller using the OpenFlow protocol. As mentioned before, POX fully supports OpenFlow 1.0 protocol and nicira extensions covering the majority of OpenFlow 1.2 protocol additions. When the controller boots up, it will listen on a socket for any incoming connections from the switches. Unlike our datapath infrastructure, which operates through the non-IP communication, the switches connect to the controller through a socket is based on IP addresses. In our case, it will be the loopback IP address *'127.0.0.1'* through which the controller listens for connections on a port say., *6633*. The OpenFlow controller and the OpenFlow switch can also be made to communicate through Layer 2 protocol by converting the INET sockets into RAW sockets based on the physical interfaces through which the switches and the controller are connected. This is not shown as part of our implementation. POX maintains a pool of connection objects and gives out one connection instance for each switch that is connected to the controller. This is shown as *'Connection Manager'* in Figure 5.1 . This connection instance is responsible for sending and receiving

OpenFlow messages between the switches and the controller.

Host 1 is connected to Switch 1 and Host 2 is connected to Switch 2 in our implementation setup. Both the switches are also connected to each other. The connection between a host and a switch and, the connection between two switches are through raw sockets that use interfaces for the connection. The interfaces that are available in the demo machine, *'eth0'* and *'lo'* are utilized for this process. For extra interfaces, multiple instances of same interfaces are used. For example, to create another virtual interface over eth0, the following command can be used : *'sudo ifconfig eth0:1 134.226.XX.XXX netmask 255.255.252.0 broadcast 134.226.XX.XXX'* where the IP address and netmask have to be changed accordingly. Due to limited availability of interfaces, few interfaces are reused for the connection between different entity pairs and the packets are differentiated by an identifier that shows which connection pair the packet belongs to. Each end of a socket is designated as a *Face* for the machine from where it originates. The connection details are shown in Figure 5.2.

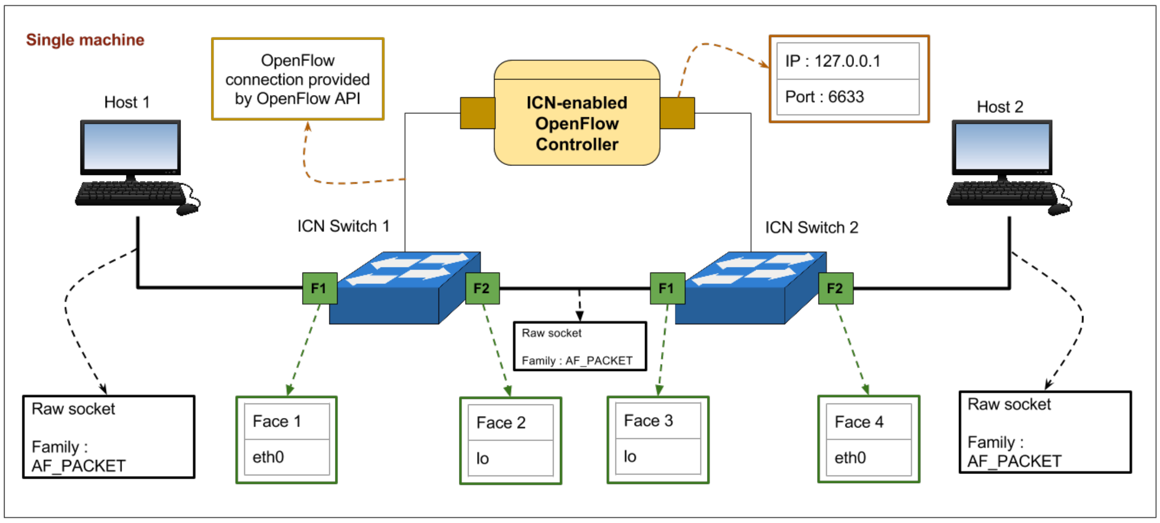


Figure 5.2: Implementation of connection between the components in OF-ICN

5.5 Raw ethernet frames

The previous section showed that how the network entities are connected in our experiment setup. This section depicts the packet structure that will be used for demonstrating the ICN capabilities in OpenFlow platform.

Existing ICN implementations operate on 'Interest' and 'Data' packets. Interest is the packet that expresses a particular user's intention to retrieve a content and the data packet is the actual content that satisfies the interest. Interest packet can be compared to a *'query'* and the data packet as a *'response'* to the query. Even though the code level implementation of interest and data packets differ between the ICN implementations, conceptually they are same. In that context, at a minimal level, interest packet should carry a content prefix and on the other hand the data packet should carry a content prefix and the corresponding data. We have exploited this minimality in our experiment such that our focus would be on making the OpenFlow start supporting the basic essential requirements of ICN.

ICN is a non-IP protocol and, it is not expected for the packets to carry the content producer's IP address in them. At any point of time, the packets can only move from one machine to another machine in the network in *hop-to-hop* fashion. This hop-to-hop transmission takes place until the packet reaches a node that provides a copy of the content. In order to support the minimal requirement of an ICN-based packet and to operate on a non-IP platform, we have used raw Ethernet frames to carry the interest and data packets between the communication entities. At any point in time, the Ethernet packet with the interest or the data will be transmitted from one machine to the next nearest one-hop machine towards the data or the requester. The packet structure, that is used in our implementation is shown in Table 5.1. This is the packet that a host sends to a switch and gets communicated between the switches as well.

Table 5.1: OF-ICN Ethernet frame structure

Field	**Description**
Ethernet Source	MAC address of the machine which is sending the packet out in the face
Ethernet Destination	MAC address of the machine which is receiving the packet. This is the machine or device that is connected to the face of source machine
Ethernet Type	Type of the payload. In our case, we send raw Ethernet payload. So it can be set to '0x7A05' or '0x0805' or '0x0801' where last two denote telecommunication units
Payload	Payload is the actual packet that is carried by the Ethernet frame. In our case, the payload contains raw content name in interest packets or, raw content name and data in data packets

5.6 Threads

After studying the packet structure in the previous section, in this section let us discuss how threading is used in our experiment for the processing of packets by the network elements.

Python 'threading' is used in our experiment to keep the hosts and switches listening for interest and data packets in separate threads. This is essential in order to make the network entities to do multiple operations at the same. While a switch is receiving an interest from a face , it can also receive a data packet from another face, for an interest which is already sent upstream towards the content. Threading is implemented as part of *'packet handler'* shown in figure 5.1. When the packet handler receives a packet, it examines the type of the packet and, hands over to the corresponding thread depends on the type of the packet. Each socket in a switch that connects to a face, runs in a separate thread in order to send and receive data packets in that face without interfering the switch operations in other faces.

Additionally, each switch and controller connection runs in a separate *IOWorker* thread which is a part of the *'Connection manager'* shown in figure 5.1 . This IOWorker thread takes care of connecting the switches and the controller in different sockets, sending message packets out in the sockets and buffering incoming message packets from the sockets. These workers allow the switches to execute other actions instead of waiting for the message to go out in the wire or waiting for the message to get buffered completely. Thus they help for the *asynchronous* message transmissions between the switches and the controller

5.7 Table data structures

In the last section, we have seen the required components for the experiment, about their connection and the packet structure supported by them. Before moving on to how the packets are processed through messaging and actions, in this section let us discuss the table structures introduced as part of this experiment and see how they are implemented.

Each switch has the following three tables as part of our experiment: FIB, PIT and CS as explained in the design chapter. These tables are implemented over Python's *List* data structure which provides essential table based operations like insert, remove , count and sort. A python class is created as a wrapper around the list for each table to provide custom operations, which other switch modules can invoke. These wrappers are illustrated as *'Table handlers'* in figure 5.1 . The table entries are also abstracted into a class such that each table instance can have any number of table entry instances. A switch creates an instance of each of these table handlers and calls corresponding functions as per the action it has to perform. A switch also maintains a *dictionary* for each table, that holds the entries that are put on the table. This is implemented to visualise the tables in UI. All the tables have a *'match' field* which is, of the type 'OFP_MATCH'. This field is used to store content prefix and helps the switch to compare a packet with the table entries. Some of the other notable fields are: FIB has a field called *'actions'* which is a list of actions of the type 'OFP_ACTION'. PIT has a field called *'faces'* which is a list of faces waiting for a content. CS table has a field called *'data'* which is a python variable to hold content.

The following code snippets of table constructors portray their implementation structures :

FIB constructor

```
#FIB constructor
class FibTableEntry (object):

  def __init__ (self, priority=OFP_DEFAULT_PRIORITY,
  idle_timeout=0,hard_timeout=0, match=ofp_match(),
  actions=[], now=None):
    """
    Constructor for FIB table entry
    """
    if now is None: now = time.time()
    self.created = now
    self.last_touched = self.created
    self.priority = priority
    self.idle_timeout = idle_timeout
    self.hard_timeout = hard_timeout
    self.match = match
    self.actions = actions
```

PIT constructor

```
#PIT constructor
class PitTableEntry(object):

    def __init__(self, priority=OFP_DEFAULT_PRIORITY,
    idle_timeout=0, hard_timeout=0, match=ofp_match(),
    faces=[], now=None):
      """
      Constructor for PIT table entry
      """
      if now is None: now = time.time()
      self.created = now
      self.last_touched = self.created
      self.priority = priority
      self.idle_timeout = idle_timeout
      self.hard_timeout = hard_timeout
      self.match = match
      self.faces = faces
```

CS constructor

```
#Content Store constructor
class ContentStoreEntry(object):

  def __init__(self, priority=OFP_DEFAULT_PRIORITY,
  idle_timeout=0, hard_timeout=0, match=ofp_match(), data="",
  now=None):
    """
    Constructor for Content Store table entry
    """
    if now is None: now = time.time()
    self.created = now
    self.last_touched = self.created
    self.priority = priority
    self.idle_timeout = idle_timeout
    self.hard_timeout = hard_timeout
    self.match = match
    self.data = data
```

5.8 Pipeline processing

The previous sections explained the packets used in the experiments and the table structures in a switch. It also showed, how they can be constructed using python scripts. This section outlines the implementation details behind the packet processing operation utilising the methods to carry out pipeline processing. This is shown as *'Pipeline Processing'* in figure 5.1 . Table 5.2 shows the knowledge of a switch in terms of major fields, which are required for the switch to process an ICN packet.

Table 5.2: OF-ICN switch state

Field	**Description**
dpid	The datapath identifier that uniquely identifies the switch in the network especially to the controller. Controller can retrieve this id from the connection object when the switch is connected to it
switch_name	The internal name for the switch which can be communicated to the controller and the hosts that are connected to the switch as an additional identifier
miss_send_len	This is the field used to decide how much information has to be sent to the controller when there is a packet miss. In order to send the entire packet to controller, this field has to be increased
fib_table	This field is used to hold the instance of FIB table for each switch. This field is initiated by using an initiation method called *'init_fib_table'*
pit_table	This field is used to hold the instance of PIT table for each switch. This field is initiated by using an initiation method called *'init_pit_table'*
content_store	This field is used to hold the instance of the content store cache for each switch. This field is initiated by using an initiation method called *'init_content_store'*
faces	This is the list of datapath connections the switch has through a number of faces. In our implementation, this list stores the sockets that are connected to different faces
faces_to_dev	This field stores the identifiers of the entities that are connected to the switch through its faces. For example in our case, for switch 1, this list has the identifiers of host 1 and the switch 2, which are connected to switch 1 through its faces
ofp_handlers	This is the list of python functions that handle different OpenFlow messages received from the controller
action_handlers	This is the list of functions that can handle different actions that are invoked internally in the switch and through the OpenFlow messages received from the controller

When the switch receives a packet the Ethernet header and payload part of the packet are extracted and separated. The Ethernet header contains Ethernet protocol which indicates what type of packet the payload contains. Our experiment is handling a non-IP packet and the non-IP packets come under the Ethernet protocol *'1402'*.

The following algorithm shows the packet identification process carried out by a switch when it receives a packet in a face

Algorithm - Packet identification process

```
if ethernet_protocol == 1402 :
    payload = extract the payload from the packet
    face = incoming face in which the packet arrived
    if payload contains 'Interest' :
        extract the content prefix
        send (prefix,face) to Interest handler
    else if payload contains 'Data' :
        extract the content prefix and the content
        send (prefix,content,face) to Data handler
    else if payload contains 'Content Announcement'
        extract the content prefix
        send (prefix,face) to the Content Registration handler
```

The flow processing for the 'interest' packet is shown in Figure 5.3 . When an interest packet is received, the content prefix is extracted from the packet by the switch and an OpenFlow match object is created. This match object is used to check for entries in the switch tables involved in the pipeline processing. First, the match object is compared with entries in Content Store cache of the switch. This matching is carried out by comparing the content prefix in the match object, with the prefix a cache entry has. If the match object matches with any of the cache entries, then the corresponding content from the

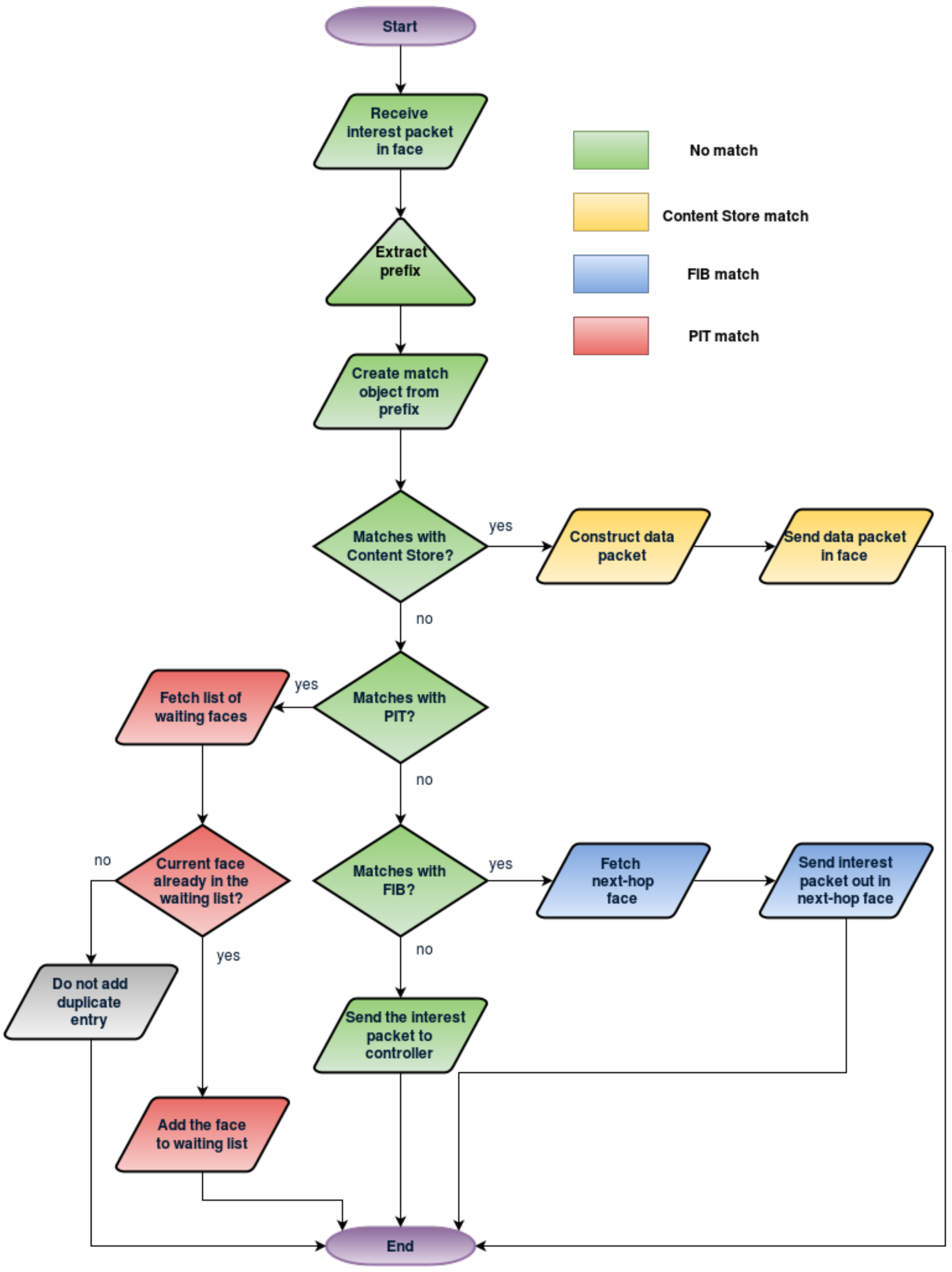

Start
Receive interest packet in face
Extract prefix
Create match object from prefix
Matches with Content Store?
yes
Construct data packet
Send data packet in face
no
Matches with PIT?
yes
Fetch list of waiting faces
no
Current face already in the waiting list?
no
Do not add duplicate entry
yes
Add the face to waiting list
Matches with FIB?
yes
Fetch next-hop face
Send interest packet out in next-hop face
no
Send the interest packet to controller
End
No match
Content Store match
FIB match
PIT match

Figure 5.3: Interest processing by OF-ICN switch

cache is retrieved and constructed into a data packet. This data packet is sent back in the incoming face from where the interest packet is received. If there is no match with the content store, then the match object is compared with the entries in Pending Interest Table. If it matches with any of the entries in PIT, then the corresponding list of waiting-faces will be retrieved. If the current face which received the interest packet is already in the list then it will not be added again. Otherwise, the incoming face will be added to the list and the entry will be updated. If the match object does not match with any of the PIT entries, then it is compared with Forwarding Information Base table. If it matches with FIB, then the corresponding list of next-hop faces will be retrieved. The interest packet will be sent in each of the next-hop faces retrieved from FIB. If none of the tables matches with the interest packet, then a PACKET_IN message is created with the content prefix and sent to the controller that is connected to the switch. The following pseudo code depicts the process outlined in this paragraph:

Algorithm - Interest Handler

```
#Input : content prefix (prefix) , face
match = create an ofp_match with the given prefix


#check 1
check in 'Content Store' table for an entry that matches the 'match'
if entry.match.prefix == prefix:
    content = entry.data
    data_packet = prefix + content
    send the data_packet back in the requested face
else :
    Goto check 2


#check 2
check in 'Pending Interest Table' for an entry that matches the 'match'
if entry.match.prefix == prefix:
    if face in entry.faces:
        face is already listed in the pending list. Do nothing
    else :
        entry.faces.append(face) #Add the face to the waiting list
else :
    Goto check 3
```

Algorithm - Interest Handler - continued

```
#check 3
check in 'Forwarding Information Base' table for an entry that matches
    the 'match'
if entry.match.prefix == prefix:
    action_list = entry.actions
    for action in action_list:
        if action == packet_out_in_face:
            Send the packet in the out face
            Delete the entry from PIT for the corresponding prefix
        else:
            execute the action
else :
    Goto check 4

#check 4
if none of the tables with the interest:
    #Send the packet to controller by creating PACKET_IN OpenFlow message
    message = ofp_packet_in(prefix,face)
    controller_connection.send(message)
```

The flow processing for a data packet is shown in Figure 5.4 . When a data packet is received, the content prefix is extracted from the packet by the switch and an OpenFlow match object is created. The match object is compared with the entries in Pending Interest Table. This is to check if any interests are waiting to be satisfied by the received content. The content prefix from the match object is compared with the prefixes stored in

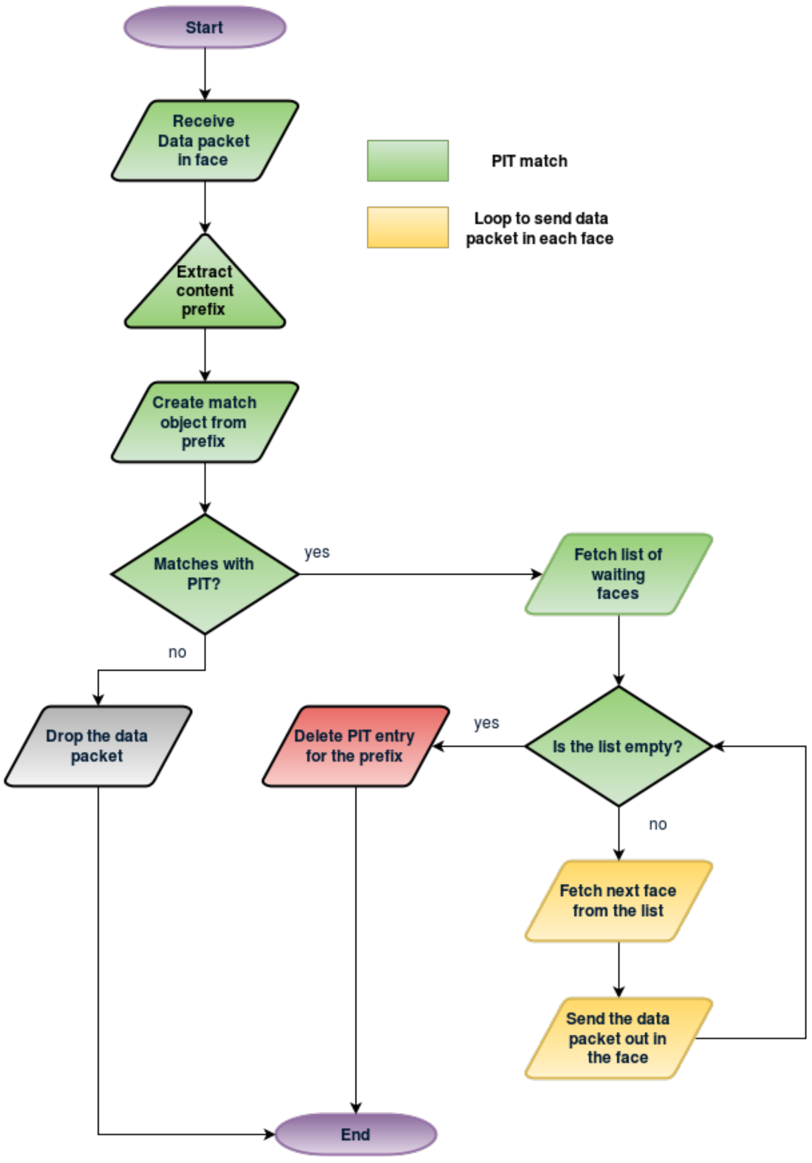

Start
Receive Data packet in face
PIT match
Loop to send data packet in each face
Extract content prefix
Create match object from prefix
Matches with PIT?
yes
Fetch list of waiting faces
no
Drop the data packet
Delete PIT entry for the prefix
yes
Is the list empty?
no
Fetch next face from the list
Send the data packet out in the face
End

Figure 5.4: Data processing by OF-ICN switch

PIT. If it matches with any of the entries in PIT, then the corresponding list of waiting-faces will be retrieved. For each face in the list, the data packet is sent out in that face towards the requester who expressed that interest. Once the data packet is sent out in all the out faces, the corresponding PIT entry is deleted from the table in order to mark that the interests are consumed by the data packet. If none of the PIT entries matches with the data packet, then the data packet is dropped assuming that the interest would have timed out or the data packet should be a duplicate one for already satisfied interest by a different upstream face. The following pseudocode depicts the process outlined in this paragraph:

Algorithm - Data Handler

```
#Input : data_packet
prefix = extract the content prefix from data_packet
match = ofp_match(prefix)

#check 1
check in 'Pending Interest Table' for an entry that matches the 'match'
if entry.match.prefix == prefix:
    waiting_faces = entry.faces
    for out_face in waiting_faces:
        #Send the data_packet in out_face
        out_face.send(data_packet)
    #Delete the entry from PIT
    delete pit_table[prefix]
else :
    Goto check 3
```

The OpenFlow switches receive content registration announcements from any of the end user hosts connected to them. Once a switch receives a content registration packet, the content prefix is extracted from the packet. An OpenFlow 'content announcement' packet is created by the switch with the extracted prefix. This OpenFlow message is sent to the controller to which the switch is connected in order to store the routing information for the content so that future interests received by any other switches for that content can be forwarded to this switch. The following pseudo code depicts the process outlined in this paragraph:

Algorithm - Content Announcement Handler

```
#Input : content_announcement_packet

prefix = extract the content prefix from content_announcement_packet
#Create OpenFlow message for content announcement
message = ofp_content_announcement(prefix)
#Send the message to controller
controller_connection.send(message)
```

5.8.1 OFP_MATCH

As shown in the algorithms in the last section, the switch creates an OFP_MATCH message, when it receives a packet with the content prefix that the packet has. This match instance is used by the packet handlers and pipeline processor to match the packet with the table entries, which is essential for the switch to decide what to do with the packet. This match structure has the fields that can be compared to the corresponding fields in the packet header. In order to support IP packets, OpenFlow introduced IP-based fields in OFP_MATCH object including IP source address, IP destination address, IP protocol and other transport layer fields like TCP source port, TCP destination port, UDP source port and UDP destination port. From OpenFlow 1.0 to OpenFlow 1.2 protocol specification, OpenFlow has undergone a lot of research on adding support to many new packet types and tags: IPV6, ICMP , MPLS tags and also to support metadata between multiple tables [6, 7]. Few of these additions are proposed as a separate addendum to OpenFlow specification and then added to OpenFlow specification 1.2. OpenFlow is continuously revised at every specification with respect to matching fields and new matching fields corresponding to packet headers are added at each release of OpenFlow specification. Even though now OpenFlow supports matching almost all the packet headers in IP-based headers, still it is lagging in supporting Non-IP based packets, especially with arbitrary field length.

In order to finalise on a field type and the length to support content prefix in OpenFlow match, ICN packet format has to be formalised. The truth is that there are many implementations of ICN in the market and each of them started following a packet structure that is suitable for that implementation. When we take ICN approach generally, the packet naming and formalisation are still under active research. We stood one step back and decided to use a general match variable for the content prefix that will in future matches with any formalised content prefix field. So, in our case, the prefix match is a logical field instead of relying on a particular name type or length.

Our flow match structure is shown below:

```
ofp_match_data = {
  'in_port' : (0, OFPFW_IN_PORT),
  'dl_src' : (EMPTY_ETH, OFPFW_DL_SRC),
  'dl_dst' : (EMPTY_ETH, OFPFW_DL_DST),
  '''
  other fields
  '''
  'interest_name':("",OFPFW_PREFIX)    #interest_name is the content prefix
}
```

5.9 OpenFlow library

OpenFlow library defines the OpenFlow protocol data structures, message and action structures. This section describes the changes that are applied to OpenFlow protocol to support ICN based messages and actions.

5.9.1 Adding a message

OpenFlow messages play important role in OpenFlow protocol as they define the communication between the OpenFlow switch and the controller and, they carry all the relevant information between them: packets, flow rules and actions. There are some OpenFlow messages, that only the switch sends to the controller. There are some OpenFlow messages, that only the controller can send to the switch. Additionally, there are some OpenFlow messages, that both switch and controller can send. These messages are differentiated in POX using corresponding Python decorators. *'openflow_s_message', 'openflow_c_message' and 'openflow_sc_message'* are the message decorators corresponding to switch-only, controller-only and both switch-controller messages. The messages are further categorized as *symmetric* and *asymmetric* messages. The messages which require an immediate reply from the destination entity is called symmetric and the fire-and-forget messages are called asymmetric messages. Each OpenFlow message is also assigned a *ofp_type* value which is a unique identifier for the message and helps to resolve the message type at the receiving end. In order to make OpenFlow support ICN, few new messages have to be defined in the library. The following steps are carried out to define a new message type in OpenFlow library using POX:

1. Created a message class in OpenFlow library file with a constant (ofp_type) and corresponding message decorator as per the need of the message

```
@openflow_s_message("OFPT_CS_FULL", 22)
class ofp_cs_full (ofp_header):
  _MIN_LENGTH = 18
```

2. For a message which the switch could send to the controller: In switch module, created a function to send that particular OpenFlow message to the controller. This function can be called anywhere from the switch module by passing the respective message object to it.

```
def send_cs_full (self):
  """
  Send CS_FULL
  """
  msg = ofp_cs_full()
  self.send(msg)
```

3. For a message from the controller which the switch could handle :Added a handler for the message (example, _rx_fib_mod) in switch class which handles when a particular message is received from the controller. These handlers are denoted as *'Message Handler'* in figure 5.1

```
def _rx_fib_mod (self, ofp, connection):
  """
  Handles flow mods
  """
  match = of.ofp_match(interest_name = ofp.interest_name)
```

New and extended messages

- The following list is the list of messages created or extended as part of our implementation using the procedure outlined above. Their functionality details are explained in the design chapter.

- OFPT_CONTENT_ANNOUNCEMENT
- OFPT_DATA_FROM_CONTROLLER_CACHE
- OFPT_CLEAR_CS
- OFPT_CS_FULL
- OFPT_FIB_FULL
- OFPT_FIB_MOD
- OFPT_FIB_REPLY
- OFPT_ADD_CS_ENTRY
- OFPT_FIB_MOD_NOTIFICATION
- OFPT_FEATURES_REPLY
- OFPT_FEATURES_REQUEST
- OFPT_DELETE_LEAST_FIB
- OFPT_PACKET_IN

5.9.2 Binary encoding

All the messages in OpenFlow are communicated through a secure channel [3]. This is enhanced by encoding the messages at the sender side and decoding them at the receiver side. OpenFlow library provides the 'packing' and 'unpacking' capability for all the OpenFlow messages. When a particular message is ready to be sent on the wire, it will be packed by the OpenFlow library in binary format and sent out on the connection. On the receiving end, the message that is received on the wire will be sent to the unpacker, which unpacks the message from binary format back to the format it is originally created. Both the switch and the controller will be involved in packing and unpacking the messages as they both involve in sending and receiving OpenFlow messages. In the network level, all the packets are interpreted as a sequence of bytes. So, the *'packing'* operation takes a list of values and convert them into a string of bytes and the *'unpacking'* operation takes a byte string and converts it back into a list of values. Packing the entire message is very important as a single byte miss might result in receiver interpreting the message differently or completely ignoring the message. Python has a package called *'struct'* which provides basic packing and unpacking functionalities. All OpenFlow message classes should have a *'pack'* method, which is called before sending out a packet. This method should be modified for each message based on the fields the message contains. 'Struct' package's pack method is called from this method by passing the message fields and the corresponding formats (like short integer, long integer , float). Raw fields need not be sent to pack method. They can be just added to the data that has to be sent in the wire. Similarly, every OpenFlow message class should have a *unpack* method that converts the byte string back into individual message fields

The following code snippet shows how an FIB_MOD message is packed before being sent in the wire and unpacked after being received from the wire:

```
def pack(self):
  packed = b""
  packed += ofp_header.pack(self)
  packed += struct.pack("!H", self.face)
  packed += self.interest_name
  return packed


def unpack(self, raw, offset=0):
  offset, length = self._unpack_header(raw, offset)
  offset, self.face = _unpack("!H", raw, offset)
  offset, self.interest_name = _read(raw, offset, length - 10)
  assert length == len(self)
  return offset, length
```

5.9.3 Adding an action

The OpenFlow messages from the control plane are usually treated as commands to datapath layer and executed by the switches in the datapath. OpenFlow messages from controller can result in any one of the following:

1. Add a rule in any of the switch tables
2. Instruct the switch to do an action
3. Instruct the switch to send a packet out in a face
4. Change the switch status by modifying the switch knowledge

OpenFlow library defines all the actions that are supported by the OpenFlow switches along with a constant to uniquely identify each action. Unlike OpenFlow messages, POX has a single python decorator for the actions called *'openflow_action'*, as all the actions are executed in the datapath. In addition to existing OpenFlow actions, few actions are added newly for the switch to support ICN functionalities and new tables. In order to add an action, the following steps are carried out:

1. Created a python class for the action with the decorator 'openflow_action'
2. Added a handler for the action (example, _action_add_pit) in switch class, that handles when a particular action is invoked in the switch. These handlers are represented as *'Action handlers'* in Figure 5.1

   ```
   @openflow_action('OFPAT_ADDPIT', 900)
   class ofp_action_addpit (ofp_action_base):
     def __init__ (self, **kw):
   ```

3. In order to send an action in a message, controller had to build an object for the action using its constructor and pass it to the switch by encapsulating in an OpenFlow message

```
action = of.ofp_action_outputface(face=face)
msg.actions.append(action)
```

4. Switch could internally invoke an action by creating an action object and handing it over to the respective handler (for example, _action_addpit)

```
def _action_addpit(self, action, packet, in_port):
  match=ofp_match(interest_name=action.interest_name)
  self.pit_table.add_entry(PitTableEntry(match=match,
   ports=action.ports))
```

New and extended actions

- The following list is the list of actions that are created or extended in our implementation using the procedure outlined above. Their functionality details are explained in the design chapter.
  - OFPAT_ADDPIT
  - OFPAT_REMOVE_PIT
  - OFPAT_OUTPUTFACE
  - OFPAT_CLEAR_CS
  - OFPAT_DELETE_LEAST_FIB

5.9.4 Adding an event and event handler

POX works in *'publish and subscribe'* methodology for handling incoming OpenFlow messages. A POX component can fire an event(in other words, publish the event) and, another POX component can catch or listen for the event(in other words, subscribe to the event). An OpenFlow message, a controller receives will be given to a message handler. This mapping is realised as follows:

```
handlerMap = {
  of.OFPT_HELLO : handle_HELLO,
  of.OFPT_ECHO_REQUEST : handle_ECHO_REQUEST,
  of.OFPT_ECHO_REPLY : handle_ECHO_REPLY,
  ......
  ......
  #New handlers for new messages
  of.OFPT_CS_FULL : handle_CS_FULL,
  of.OFPT_CONTENT_ANNOUNCEMENT : handle_CONTENT_ANNOUNCEMENT,
```

The message handler will extract the message type and convert the message into an event under the connection object, which maintains the connection between the switch and the controller. Once the event object is created in the connection object, it will be raised by the object and the event will be invoked. The component which is listening for that event will catch the event and execute the handler code. These main functions are shown as *'Event Trigger'* and *'Event Handler'* at controller side in Figure 5.1 . The event handlers can then invoke the modules, that are responsible for other operations; maintaining the routing table or managing the internal cache.

The following steps are carried out to add an event and event handler to the OpenFlow controller:

1. Created an event class with a unique name (for example, CsFull) by inheriting the parent *Event* class of POX.

   ```
   class CsFull(Event):
     """
     Event raised when the Content Store is Full
     """
     def __init__(self, connection):
       Event.__init__(self)
   ```

2. As OpenFlow connection object between the switch and the controller is responsible for raising the event, the class name which is created in the above step is added to the list of the events the connection object can raise. This is achieved by creating an object for *EventMixin* POX module in connection class and adding the event name to it. EventMixin gives the provision to the class which inherited it to raise an event from its list.

   ```
   class Connection (EventMixin):


   _eventMixin_events = set([
       ConnectionUp,
       ConnectionDown,
       #new events
       CsFull,
       ContentAnnouncement,
     ])
   ```

3. Created a handler for the event in the controller component to execute a piece a code as the response for the triggered event (for example, _handle_CsFull)

   ```
   def _handle_CsFull (self, event):
       '''
       Handles CS Full event
       '''
   ```

5.10 Source code

The source code of the complete project is published online in the following GitHub repository : *https://github.com/jeevarajendran/pox*

After cloning above repository into the home folder, the following commands can be used to invoke different modules involved in the experiment:

Controller :

```
$ cd pox
$ sudo ./pox.py log.level misc.controller
```

Switch 1 :

```
$ cd pox
$ sudo ./pox.py log.level datapaths:icnswitch1
```

Switch 2 :

```
$ cd pox
$ sudo ./pox.py log.level datapaths:icnswitch2
```

Host 1 :

```
$ cd pox/pox/hosts
$ sudo python host1.py
```

Host 2 :

```
$ cd pox/pox/hosts
$ sudo python host2.py
```

5.11 Summary

This chapter described the implementation details of OF-ICN. The technical architecture outlined in Figure 5.1, detailed the various components involved as part of our implementation. We use the open source Python OpenFlow controller, POX, for implementing our approach. POX provides the following OpenFlow components as separate modules: OpenFlow switch, OpenFlow controller and OpenFlow protocol library. Each of these modules are modified and supported with additional sub-modules to implement the required functionalities in modularized code blocks. The details of the connection between the components involved in our implementation, are shown in this chapter.

The following switch components are realized as part of our implementation:

- Connection manager
- OpenFlow message handler
- Action handler
- Pipeline processor
- Table handler
- Packet handler.

The OpenFlow switch is modified to incorporate the ICN-based table structures, which include, Forwarding Information Base, Pending Interest Table and Content Store. These structures are implemented using Python's 'List' data structure and corresponding Python dictionaries are developed to query and verify in UI. OFP_MATCH field, which is a part of all the table structures, is explained.

The following controller components are implemented:

- Connection manager
- OpenFlow message handler
- Event trigger
- Event Handler
- Cache manager
- Routing DB manager (Topology manager)

This chapter also detailed the steps to add a new message, an action, an event or an event handler to the OpenFlow protocol library and, the changes that should be applied to the supporting modules are presented. The Ethernet packet frame which is used to construct the interest and data packets are outlined along with its fields. The test environment is described along with the necessary packages and the installation steps. The commands to execute the experiment code from GitHub are listed before this section.

Chapter 6

Evaluation and Discussion

The main motive of this study is to report the feasibility of OpenFlow to provide ICN functionalities by creating a control plane for ICN using OpenFlow. As mentioned in the earlier chapters, even though there are multiple implementations of ICN; NDN, CCN and NetInf, they all almost provide similar features to furnish network communication though named contents. Each implementation has its own supporting tools as well. For instance, most of them support, request forwarding for named contents and provisions for node-level caching. Keeping the generalisation in mind, instead of using a particular implementation of ICN while modifying OpenFlow, we have formulated the solutions based on the generic architecture of ICN. Moreover, the underlying aim of this study is to bring together the advantages of both OpenFlow and ICN technologies so that one can be benefited by other. Thus, Basic features of both OpenFlow and ICN are selected as the benchmarking metrics for the implementation. This section starts by listing the categories considered for the evaluation process, followed by the test cases utilised to verify the categories and the discussion on outcomes.

6.1 Categories

A hidden responsibility of this whole study is to preserve the working features of OpenFlow which made it be one of most widely used SDN architectures but still, enable it to support ICN. The modifications and extensions as part of this implementation should abide those features. In order to evaluate our objectives, we have divided the evaluation process into three categories: They are:

- Category 1 : Ensure that the modified OpenFlow affords ICN functionalities
- Category 2 : Ensure that the basic features of OpenFlow are still preserved
- Category 3 : Ensure that the implementation is easy to deploy
- Category 4 : Ensure that the modified OpenFlow provides advantages over existing ICN implementations

These categories are used to benchmark our goals. The following sections present the items which are kept as marking scales under each of these categories and depicts how well our modified OpenFlow can support those items.

6.2 Category 1 : ICN functionalities over OpenFlow

The following section outlines the benchmarking points that are used to ensure that the modified OpenFlow provides the necessary functionalities of ICN and their outcomes:

6.2.1 Labeling content 'vs' end to end addressing

The very basic principle for any ICN framework is to shift the addressing paradigm from *'host'* to *'content'*. Each and every piece of information should be labelled and addressable in the ICN network. As per OpenFlow, at every decision-making point, the endpoint address is also taken into consideration in order to help the packet reach its destination

Table 6.1: Evaluating the presence of content name in OF-ICN

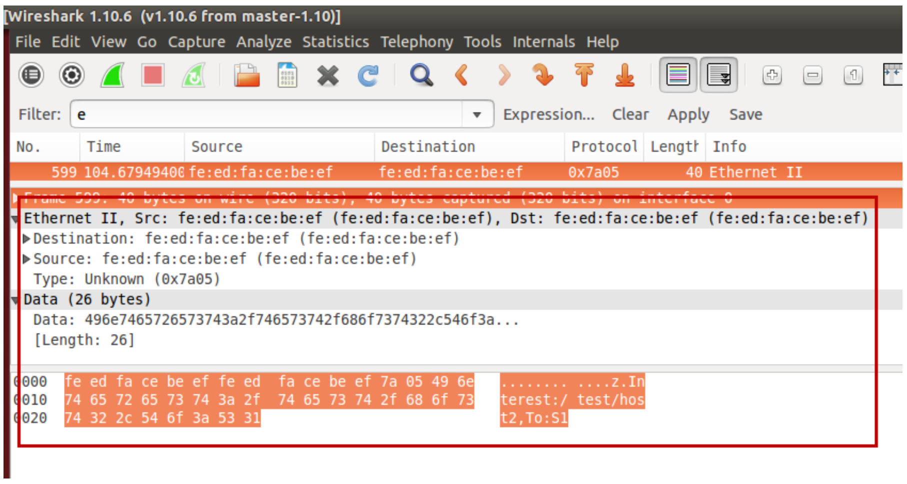
Test Scenario	**Methodology**	**Result**
Test that all the packets that pass through the datapath addresses content instead of end host	Wireshark packet sniffer • Start the Wireshark • Filter with the Ethernet protocol • Capture the packets that are put by the network elements in the interfaces under inspection • Analyse the packets	Ensured that the Interest and Data packets which are communicated between the switches and the hosts, carry the content name and nowhere they address the destination end host IP address

end-point. Our implementation has altered this to support ICN so that the content name is considered as a primitive for decision making. For the matter of simplicity, we have used a *'flat'* human readable naming scheme for the data. In order to verify that our ICN implementation works with named contents at any point of time, we have used *'wireshark'* to verify the packets which pass through our datapath implementation and ensured that they address the request or the content by its name instead of using an IP address to address the end point.

Table 6.1 and Figure 6.1 show the test scenario carried out to evaluate this section

6.2.2 Requests aggregation

'Request aggregation' is the process in which multiple interests for the same content will be aggregated and collided in a switch that receives those interests [55]. Only the first interest in that pool will be out on the network looking for the content. These multiple requests can be from the same machine requesting the same content more than once or, it can be from multiple hosts which are interested in the same content. In either case, the switch that receives those interests should recognise them as duplicate interests for the one which is sent upstream and aggregate them into a list . This process helps to improve the performance of the network by not flooding with duplicate requests. Our implementation

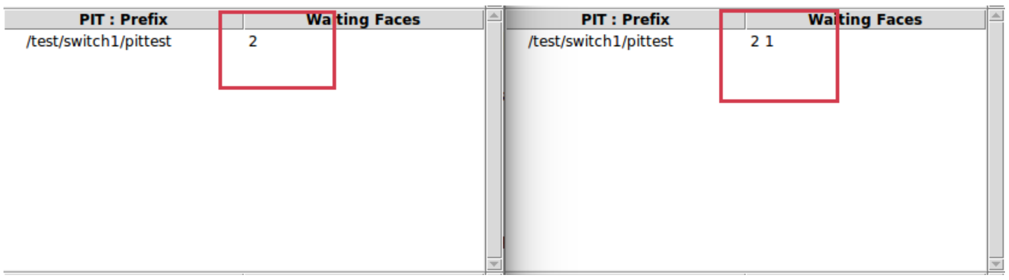

[Wireshark 1.10.6 (v1.10.6 from master-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: e
Expression... Clear Apply Save

```
No.   Time          Source              Destination         Protocol  Length Info
  599 104.67949400 fe:ed:fa:ce:be:ef  fe:ed:fa:ce:be:ef   0x7a05        40 Ethernet II

Frame 599: 40 bytes on wire (320 bits), 40 bytes captured (320 bits) on interface 0
Ethernet II, Src: fe:ed:fa:ce:be:ef (fe:ed:fa:ce:be:ef), Dst: fe:ed:fa:ce:be:ef (fe:ed:fa:ce:be:ef)
  Destination: fe:ed:fa:ce:be:ef (fe:ed:fa:ce:be:ef)
  Source: fe:ed:fa:ce:be:ef (fe:ed:fa:ce:be:ef)
  Type: Unknown (0x7a05)
Data (26 bytes)
  Data: 496e7465726573743a2f746573742f686f7374322c546f3a...
  [Length: 26]

0000  fe ed fa ce be ef fe ed  fa ce be ef 7a 05 49 6e   ........ ....z.In
0010  74 65 72 65 73 74 3a 2f  74 65 73 74 2f 68 6f 73   terest:/ test/hos
0020  74 32 2c 54 6f 3a 53 31                            t2,To:S1
```

Figure 6.1: Labeling content : Snapshot from Wireshark showing the content name in the packet and the packet is not addressed with the end-point IP address

provides an option to do such interests aggregation by exploiting the *'Pending Interest Table'* which is one of the switch's flow processing tables.

Table 6.2 shows the test scenario carried out to evaluate this section and Figure 6.2 shows the results.



PIT : Prefix	Waiting Faces
/test/switch1/pittest	2

PIT : Prefix	Waiting Faces
/test/switch1/pittest	2 1

Figure 6.2: Evaluating request aggregation : The faces that requested for the same content are aggregated in a list in PIT

6.2.3 Multicasting

OpenFlow does not support multicasting for IP packets expect there is an overlay network to provide it [102]. At any point of time, a packet is sent out through a single flow towards the destination endpoint. This is expected in a network where there is no in-network storage of the content and only the destination end-point has the content. Whereas, in

Table 6.2: Evaluating request aggregation procedure

Test Scenario	**Methodology**	**Result**
Test that multiple requests for the same content from same host or different hosts are getting aggregated in the ingress switch	Pending Interest Table • **Topology :** Connect two hosts H1 and H2 to Switch S1 • I1 : From H1 send an interest packet for the content name *'/test/h3/video1'* produced by host H3 • I2 : From H2 send an interest packet for the same content name *'/test/h3/video1'* • Check the PIT table in Switch S1 using the UI viewer *'tkinter'* • Check the number of interests being received by host H3	Ensured that the interests I1 and I2 are aggregated in Switch's PIT and only the first interest I1 reaches the producer H3

ICN framework, when there is a provision for in-network caching in the network elements, it suggests multicasting for ICN packets. Thus, an interest packet can be sent in multiple faces out from a switch looking for a possible storage of a content. The *'Forwarding Information Base'* table in our implementation helps to achieve this multicasting by allowing multiple actions to be executed for a packet match and, each action can send out the packet in a different face towards the copy of the content.

Table 6.3 shows the test scenario carried out to evaluate this section.

6.2.4 Scalability

Scalability is a big research area which is being considered separately by many research studies and it has its implications in both the technologies, OpenFlow and ICN [103, 22, 32, 26]. Our implementation objective is to provide the scalability in terms of a number of switches and hosts that can be connected to the network. Being an OpenFlow switch, our switch can be connected to a number of hosts which depends on the number of faces

Table 6.3: Evaluating OF-ICN multicasting functionality

Test Scenario	**Methodology**	**Result**
Test that the modified OpenFlow switch can send out interest packets in multiple faces	Forwarding Information Base • **Topology :** Consider three switches S1, S2, S3 connected to controller • S1 is connected to S2 and S3 through two different faces F2, F3 • Connect a host H1 to S1 and send out an interest packet named 'test/h3/video1' • Push flow rule from controller such that the action list contains both the faces F2 and F3 of the switch S1 as output faces • Ensure that both the switch S2 and S3 receive the interest packet by sniffing the interfaces through Wireshark packet sniffer	Ensured that the interest is multicasted in both the faces F2 and F3 of switch S1 and reaches both the switches S2 and S3

available for the switch. Similarly, a switch can be connected to any number of switches as well through its interface abstraction, *faces*. The OpenFlow controller can handle multiple switches, as it hands over each connection to a worker by creating a logical instance of each switch.

The other direction of scalability is to support the increasing number of content names and how well the switches can handle them in their forwarding tables. OF-ICN controller has a superset of forwarding information of the switches so that popular routes alone gets stored in the FIB of the switches.

6.2.5 Compatibility with IP-packets

One of our responsibilities during the implementation is to provide the new extensions as an addendum to the existing OpenFlow specification, thus, OpenFlow can still provide IP-based flow control and on top of it, it can control ICN based flows as well. In order to achieve this, we have modularized the ICN functionalities such that they do not interfere with IP-based communications provided by OpenFlow. The OpenFlow switches and controller are equipped with the modules to differentiate the packets and, hand over to the handlers based on the packet types. For now, we have the pipeline processing for ICN, that is separated from IP pipeline processing in order to provide the cordiality between both operations. In future, we expect to integrate them into one pipeline process and, provide better mutuality.

Table 6.4 shows the test scenario carried out to evaluate this section.

6.2.6 Flow rule based advantages

Every first unrecognised packet by a switch is expected to be sent out to the controller for routing resolution. The OpenFlow controller can be a local controller or a remote controller. In either case, it will impose a delay on the packet transmission including the time, the packet takes to reach the controller, the time, the controller takes to resolve the

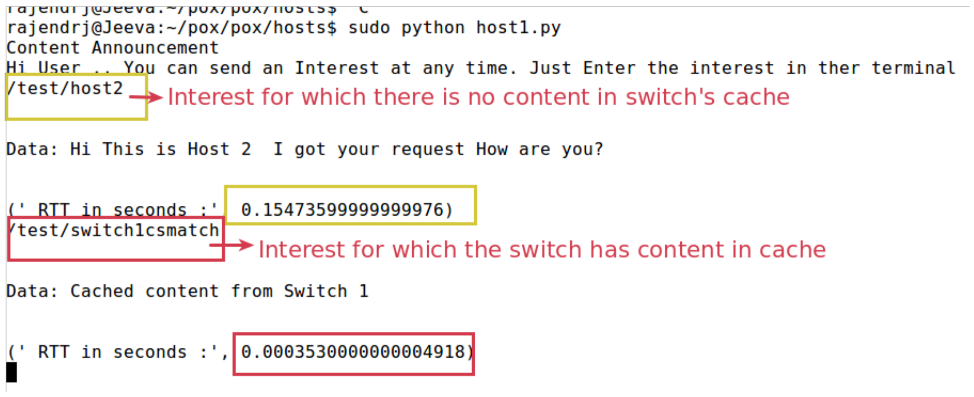
Table 6.4: Evaluating the compatibility of OF-ICN with both IP and ICN packets

Test Scenario	**Methodology**	**Result**
Test that the modified OpenFlow can handle both ICN and IP packets and forward them based on the corresponding forwarding tables	• **Topology :** Consider a topology where a switch S1 is connected to an OpenFlow controller. Hosts H1 and H2 are also connected to switch S1. H2 produces a content named '/test/h2/file1' and it has an IP address assigned to it. Assume that the switch has details about both the hosts and the contents they produce • Send an IP packet keeping H2's IP address as destination address • Send an ICN interest packet for the content name 'test/h2/file1' • Ensure that both the packets reach Host H2	Ensured that both the IP packet, which is designated to H2's IP address and, the ICN packet, which is sent out for the content produced by H2, reach the host H2

packet, the time takes for the flow rules to be pushed to switch and, finally, the time takes to send out the packet based on the action in the pushed flow rule. Once a flow rule is pushed down to a switch, further packets that satisfy the same flow rule need not be sent to the controller, which significantly reduce the overall transmission time for the packet. Table 6.5 shows the test execution for this scenario.

6.2.7 In-network caching

Current OpenFlow switches can exploit their internal memory to store flow rules for IP-based flows [21]. Similarly, our proposed ICN-enabled OpenFlow switches are modified to utilise their memory to store contents. This is achieved by storing the data packets that are received for the interests. This caching depends on the capacity of the switch and can be extended by utilising extended memory. In our implementation, we have shown how this in-network caching can be realised using the *'Content Store'* option. We have evaluated our implementation to show the time, that is taken to retrieve a content which is cached in an intermediate switch, will be less than the time taken to retrieve the data from the content producer, which is shown in Table 6.6 and Figure 6.3



```
rajendrj@Jeeva:~/pox/pox/hosts$ ^C
rajendrj@Jeeva:~/pox/pox/hosts$ sudo python host1.py
Content Announcement
Hi User .. You can send an Interest at any time. Just Enter the interest in ther terminal
/test/host2

Data: Hi This is Host 2  I got your request How are you?

(' RTT in seconds :', 0.15473599999999976)
/test/switch1csmatch

Data: Cached content from Switch 1

(' RTT in seconds :', 0.0003530000000004918)
```

Interest for which there is no content in switch's cache
Interest for which the switch has content in cache

Figure 6.3: In-network caching : The time taken to receive a content from a cache is lesser than the time taken to receive it from the end-point producer

Table 6.5: Evaluating the time difference between a controller flow and a flow based on the flow rule in switch

Test Scenario	**Methodology**	**Result**
Test that the time taken for an ICN communication with flow rule is lesser than the time taken for an ICN communication without a flow rule in the switch	• **Topology :** Consider a topology where a switch S1 is connected to an OpenFlow controller. Hosts H1 and H2 are also connected to switch S1. H2 produces a content named '/test/h2/file1'. Assume that the switch has details about both the hosts and the contents they produce • **Iteration 1 :** Send out an interest packet for the content '/test/h2/file' from H1 • **Time 1 :** Measure the time taken for the host to receive the data • **Iteration 2 :** Send out another interest packet for the content '/test/h2/file' from H1 • **Time 2 :** Measure the time taken for the host to receive the data • Ensure that Time 2 is lesser than Time 1	Ensured that the time taken for a 2nd interest packet for the same content received through 1st interest, is lesser due to the availability of flow rule in the switch and the packet is not sent to the controller for resolution

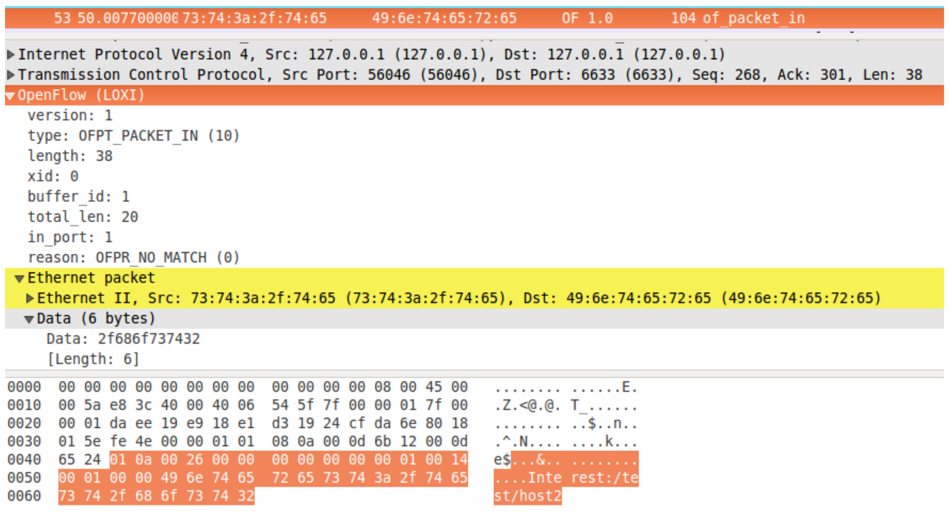
Table 6.6: Evaluating the time difference between a cache flow and cache-less flow

Test Scenario	**Methodology**	**Result**
Test that the time taken for an ICN communication for a content cached in an intermediate switch is lesser than the time taken to a content which is not cached and has to be retrieved from the content producer	• **Topology :** Consider a topology where a switch S1 is connected to an OpenFlow controller. Hosts H1 and H2 are also connected to switch S1. H2 produces two contents; **C1 :** '/test/h2/file1' and **C2 :** /test/h2/image1. Consider that C1 is cached in the switch S1 and C2 is not cached • Send out an interest packet for the content C1,'/test/h2/file1', from H1 • **Time 1 :** Measure the time taken for the host to receive the data • Send out another interest packet for the content C2,'/test/h2/image1', from H1 • **Time 2 :** Measure the time taken for the host to receive the data • Ensure that Time 1 is lesser than Time 2	Ensured that the time taken to retrieve C1 is lesser than C2, as C1 is cached in Switch S1 and C2 is not cached

6.2.8 Security

ICN frameworks, when they suggest decoupling the data from the location, take out the provision of securing the location [4]. This results in the need for the content to carry security features along with them. In other terms, the security is provided at the content level. Whereas, in OpenFlow, the communication between the switch and controller takes place through the secure channel. This is achieved by converting all the messages into binary format and make them travel through the secure channel [22]. Our implementation exploits this provision of OpenFlow and, all the new messages and actions are converted into the binary format before being sent out on the wire. This is ensured by sniffing the OpenFlow packets using Wireshark packet sniffer.

Figure 6.4 shows the snapshot of Wireshark showing the OpenFlow message 'PACKET_IN' in wire format:



```
     53 50.00770000073:74:3a:2f:74:65      49:6e:74:65:72:65      OF 1.0        104 of_packet_in

▶Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▶Transmission Control Protocol, Src Port: 56046 (56046), Dst Port: 6633 (6633), Seq: 268, Ack: 301, Len: 38
▼OpenFlow (LOXI)
   version: 1
   type: OFPT_PACKET_IN (10)
   length: 38
   xid: 0
   buffer_id: 1
   total_len: 20
   in_port: 1
   reason: OFPR_NO_MATCH (0)
 ▼Ethernet packet
  ▶Ethernet II, Src: 73:74:3a:2f:74:65 (73:74:3a:2f:74:65), Dst: 49:6e:74:65:72:65 (49:6e:74:65:72:65)
  ▼Data (6 bytes)
     Data: 2f686f737432
     [Length: 6]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 5a e8 3c 40 00 40 06  54 5f 7f 00 00 01 7f 00   .Z.<@.@. T_......
0020  00 01 da ee 19 e9 18 e1  d3 19 24 cf da 6e 80 18   ........ ..$..n..
0030  01 5e fe 4e 00 00 01 01  08 0a 00 0d 6b 12 00 0d   .^.N.... ....k...
0040  65 24 01 0a 00 26 00 00  00 00 00 00 00 01 00 14   e$...&.. ........
0050  00 01 00 00 49 6e 74 65  72 65 73 74 3a 2f 74 65   ....Inte rest:/te
0060  73 74 2f 68 6f 73 74 32                            st/host2
```

Figure 6.4: Security : New OpenFlow messages are encoded in binary format

6.2.9 Transport

'Flow balancing' is very important in a network that involves a lot of traffic or during a popular content broadcast [104]. In an IP network, where each request by a host to reach another host is treated as an individual end to end communication, providing this flow

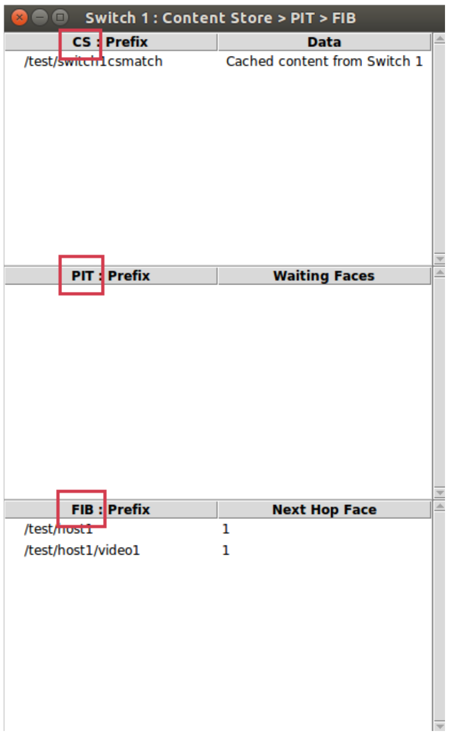
Table 6.7: Evaluating flow balancing with duplicate data packets

Test Scenario	**Methodology**	**Result**
Test that the duplicate data packets are dropped out in a switch and the consumer receives only single data packet for an interest	• **Topology :** Consider a topology where three OpenFlow switches S1, S2 and S3 are connected to an OpenFlow controller. S1 is connected to S2 and S3 through faces F2 and F3. S2 and S3 both have the content C1: '/test/h2/video1' in their caches. S1's FIB has the entry for '/test/h2/video1' with both faces F2 and F3 listed as next hop faces. H1 is connected to S1 • Send out an interest packet for the content C1, from H1 • Ensure that the host receives only one data packet for the interest	Ensured that H1 receives only one data packet in response to the interest packet even though the interest is sent out in both the faces and the data packets are arrive in both the faces

balance is very difficult [3]. On the other hand, ICN claims to achieve this flow balance in a content-centric network in two ways. One way is to enable a switch to do interest aggregation. Another way is to provide flow balance over data packets. As mentioned early, an interest packet can be sent out in multiple faces. This may result in the same data packet from different sources which have the same copy of the data and, they arrive in different times. Once the first data packet is sent out in the waiting faces, the PIT entry for the interest will be deleted from the table. Thus, for any duplicate data packets that the switch receives from any other faces, there will not be any matching entry in PIT and thus they will be dropped by the switch. This check helps to avoid duplicate packets to wander around in the network. This functionality is evaluated through the test case depicted in Table 6.7.

6.2.10 Multiple tables support

OpenFlow provided multiple table support from version 1.2 [7]. Even though POX basically built upon OpenFlow version 1.0 , it supports the Nicira and nexus extensions which provide the features of OpenFlow 1.2 and above. Exploiting these extensions of POX, our implementation provides support for multiple tables to realise the necessary ICN data structures: Forwarding Information Base, Pending Interests Table and Content Store. The 'pipeline processing' and 'Table handlers' use these tables to process the ICN packets. In our implementation, the tables are initialized by the switch when it is connected to the network. Figure 6.5 from 'Tkinter' shows that the entries in different tables as initialized by a switch when it connects to controller:



Switch 1 : Content Store > PIT > FIB

CS : Prefix	Data
/test/switch1csmatch	Cached content from Switch 1

PIT : Prefix	Waiting Faces

FIB : Prefix	Next Hop Face
/test/host1	1
/test/host1/video1	1

Figure 6.5: ICN-based multiple tables in OF-ICN switch: CS, PIT, FIB

6.3 Routing based operations

This section presents the evaluation details for some of the routing-related ICN functionalities that come under *'Category 1'* :

6.3.1 Forward-by-name

'Forward-by-name' is the process in which the packet forwarding operation is carried out using the name contained in the packet, instead of exploiting the source and destination addresses of the packet [37]. This is evaluated by sending out an interest packet using Ethernet protocol, with the content name to identify the content and, showing that, only if the switch or the controller has the content name stored in their forwarding or routing table, the packet will be forwarded. Otherwise, the packet will be dropped. The evaluation process is explained in the test case in Table 6.8.

6.3.2 FIB management

'Forwarding Information Base' of the switches that are connected to the controller is managed by the controller through OpenFlow interface. One particular FIB management algorithm that is implemented in our implementation is to send an instruction from the controller to switch, to delete the least used FIB entry when the switch's FIB is full. When the switch notifies the controller when the FIB is filled, the controller instructs the switch to delete the least used entry by sending out 'DELETE_LEAST_FIB' message. Switch identifies the least used FIB entry by querying for the FIB entry with the lowest counter value. The test case in Table 6.9 depicts how this feature is evaluated in our implementation.

Table 6.8: Evaluating 'Forward-by-name' operation

Test Scenario	Methodology	Result
Test that only the interest packet whose name can be matched with FIB of the switch or the routing table controller can be forwarded further. Otherwise, the packet should be dropped	• **Topology :** Consider a topology where a switch S1 is connected to the controller. H1 and H2 are connected to S1. H2 produces a two contents: C1 : '/test/h2/video1', C2: '/test/h2/image1'. The switch has the forwarding information for C1 based on its name. Neither Switch nor the controller has the forwarding information for C2 • Send out an interest packet for the content C1, from H1 • Ensure that the host receives the data packet from switch • Send out an interest packet for the content C2, from H1 • Ensure that the switch sends back NOACK packet to the host	Ensured that H1 receives the data packet for the interest for which the switch has matching content name in FIB. Ensured that for the interest where there is matching content name in neither switch's FIB nor controller cache, the interest packet is dropped and NOACK information is returned

Table 6.9: Evaluating FIB management when the switch's FIB is full

Test Scenario	**Methodology**	**Result**
Test that when a switch sends FIB_FULL to controller, the lease used FIB entry is deleted from the table	• **Topology :** Consider a topology where a switch S1 is connected to the controller. H1 and H2 are connected to S1. The FIB of S1 is initialized with 5 entries and the maximum entries of FIB are set to 5. The lowest counter value in FIB is '1' for 4th entry. Controller has the forwarding information for a content C1 produced by H2 • Send out an interest packet for the content C1, from H1 • When the switch tries to add flow rule from controller for C1, Ensure that FIB_FULL message is sent from switch to controller • Ensure that the controller sends back 'DELETE_LEAST_FIB' message to switch • Ensure that the switch deletes the entry with lowest counter value (in this case, the 4th entry in FIB)	Ensured that the switch sends an FIB_FULL message to the controller and the controller sends back 'DELETE_LEAST_FIB' message, by sniffing the interface through Wireshark packet sniffer. Also ensured that the switch deletes the FIB entry with lowest counter value by analysing the FIB table before and after the FIB_FULL message, using 'Tkinter'

6.4 Content based operations

This section presents the evaluation details for some of the content-related ICN functionalities that come under *'Category 1'* :

6.4.1 Interest and Data forwarding

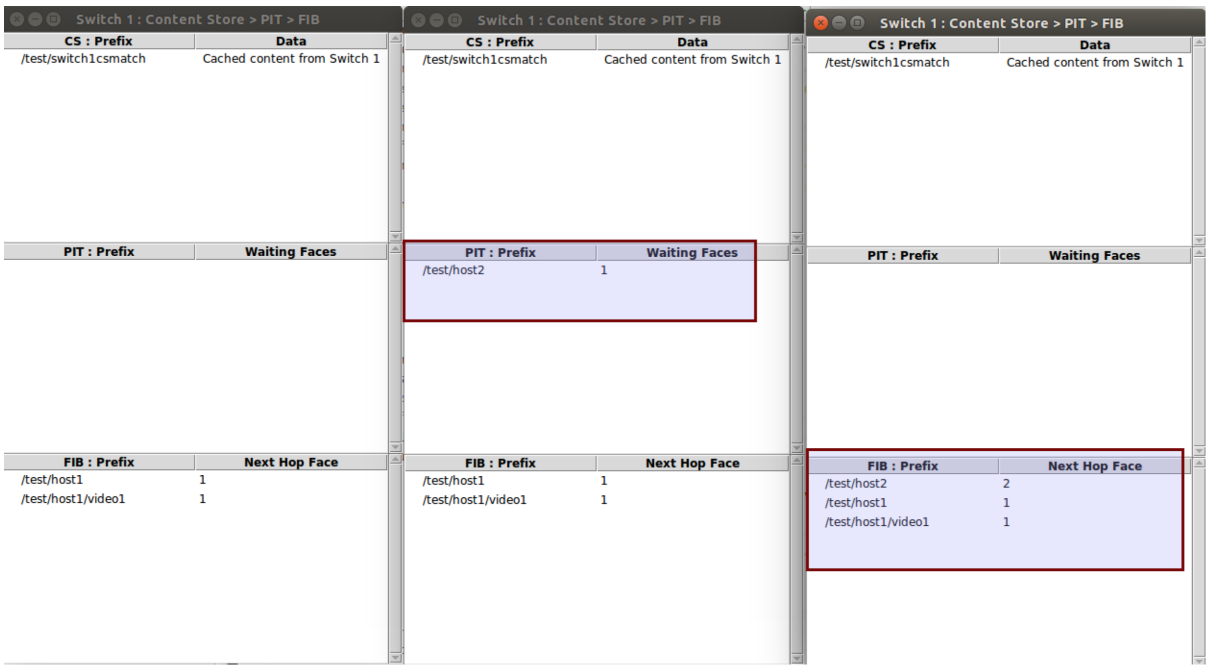
POX's switch, controller and library components are modified in our implementation to support ICN communication using interest and data packets. Using the topology outlined in Figure 4.1, the ICN-related tables in each switch are initialized with some bootstrap values. The controller's routing database and the cache are also initialized with some initial values. The process starts with sending out interest packet from one host, for the content produced by another host, connected to a different switch. The process ends when the first host receives the data packet for the interest. The next iteration is carried out by sending out interest packet from the second host and the iteration ends when the second host receives the data. Many numbers of such iterations are carried out to ensure that the methodology does not break during the process. Table 6.10 explains the process involved in one iteration of interest-data communication and Figure 6.6. shows the results.

6.4.2 Caching decisions

The switches implemented by us are not distributed. So, the OpenFlow controller, being the central controlling entity, can decide at any time what to store in the switch's cache and what to remove from the cache. For a proof of concept, we have implemented an algorithm, where the switch can inform the controller when the cache is full through 'CS_FULL' message and, the controller can give an instruction to clear the cache from the switch. The test case in Table 6.11 and Figure 6.7 illustrate this evaluation process.

Table 6.10: Evaluating the interest and data forwarding process by OF-ICN

Test Scenario	**Methodology**	**Result**
Test that a host can send out interest packets any number of times for a content and it receives back the data packet for every interest	• **Topology :** This topology is same as that of Figure 4.1 . Consider that H2 produces a content 'C' named '/test/h2/data/v1' and the Switch S2 has announced this content to the controller. Now controller has the route for 'C' • Send out an interest packet for the content C, from H1 • Ensure that Switch S1's PIT is updated with an entry for 'C' from H1 • Ensure that Switch S1's FIB is added with an entry for 'C' when the switch receives the flow table from controller • Ensure that host H2 eventually receives the interest and send back the data • Ensure that once the switch receives the data packet, it updates its content store with the name for 'C' and the received content, which may be 'C' itself • Ensure that host H1 finally receives the content • Ensure that multiple iterations of the same process work without any issues	Switch S1's table dictionaries are projected in UI after each checkpoint in the process as mentioned in 'methodology' column. Ensured that S1's PIT is updated when an interest is forwarded. Ensured that S1's FIB is updated with the forwarding rule from the controller. Ensured that S1's CS is updated with the new entry when the switch receives the content. We ensured that the host H1 receives the data for every interest packet it sends out in the network for the content 'C' until otherwise the host is down or there are no copies for the content


Switch 1 : Content Store > PIT > FIB

CS : Prefix	Data
/test/switch1csmatch	Cached content from Switch 1

PIT : Prefix	Waiting Faces

FIB : Prefix	Next Hop Face
/test/host1	1
/test/host1/video1	1

Switch 1 : Content Store > PIT > FIB

CS : Prefix	Data
/test/switch1csmatch	Cached content from Switch 1

PIT : Prefix	Waiting Faces
/test/host2	1

FIB : Prefix	Next Hop Face
/test/host1	1
/test/host1/video1	1

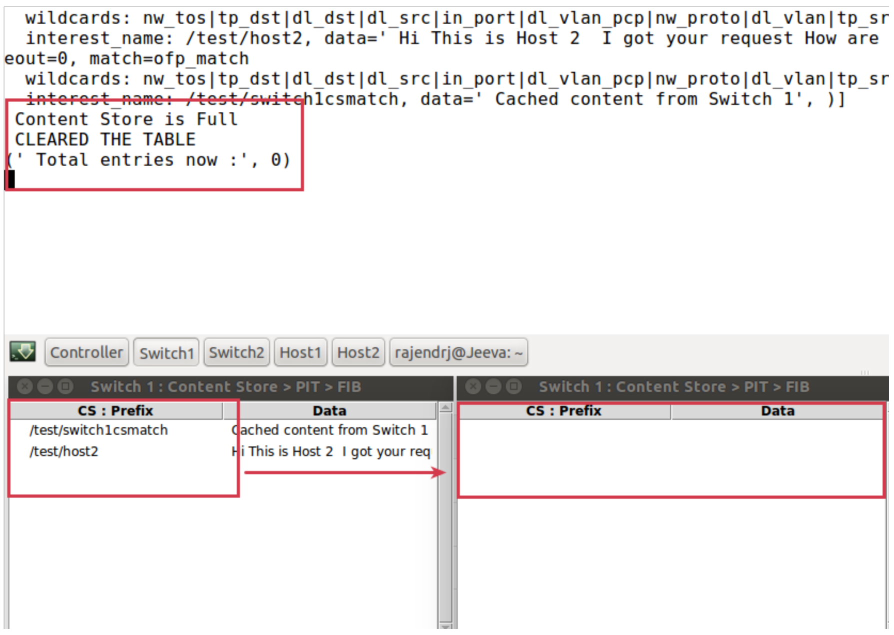
Switch 1 : Content Store > PIT > FIB

CS : Prefix	Data
/test/switch1csmatch	Cached content from Switch 1

PIT : Prefix	Waiting Faces

FIB : Prefix	Next Hop Face
/test/host2	2
/test/host1	1
/test/host1/video1	1

Figure 6.6: Evaluation of Interest processing by OF-ICN switch. Procedure in Table 6.10



```
  wildcards: nw_tos|tp_dst|dl_dst|dl_src|in_port|dl_vlan_pcp|nw_proto|dl_vlan|tp_sr
  interest_name: /test/host2, data=' Hi This is Host 2  I got your request How are
eout=0, match=ofp_match
  wildcards: nw_tos|tp_dst|dl_dst|dl_src|in_port|dl_vlan_pcp|nw_proto|dl_vlan|tp_sr
  interest_name: /test/switch1csmatch, data=' Cached content from Switch 1', )]
 Content Store is Full
 CLEARED THE TABLE
(' Total entries now :', 0)
```

CS : Prefix	Data
/test/switch1csmatch	Cached content from Switch 1
/test/host2	Hi This is Host 2 I got your req

CS : Prefix	Data

Figure 6.7: OF-ICN switch clearing the cache when it receives CLEAR_CS message from the controller

Table 6.11: Evaluating OF-ICN behaviour when the switch's cache is full

Test Scenario	**Methodology**	**Result**
Test that a host can send out interest packets any number of times for a content and it receives back the data packet for every interest	• **Topology :** This topology is same as that of Figure 4.1 . Switch S1's Content Store is bootstrapped with 5 entries and the maximum number of entries is set to 5. S1 has the route for the content 'C' produced by H2 but not yet cached in S1 • Send out an interest packet for the content C, from H1 • Ensure that host H2 eventually receives the interest and send back the data • Ensure that once the switch receives the data packet, it sends out CS_FULL message to controller • Ensure that the controller sends back 'CLEAR_CS' message to switch • Ensure that S1's content store is cleared after receiving the instruction from the controller	Wireshark is used to ensure that the switch sends 'CS_FULL' message and, the controller sends back 'CLEAR_CS' message. Switch S1's table dictionaries are projected in UI after each checkpoint in the process as mentioned in 'methodology' column. Ensured that the content store is filled with 5 entries when the switch is initialized and cleared completely upon receiving CLEAR_CS instruction from the controller

6.4.3 Caching notifications

When a host connects to a switch, it is expected that the host announce the contents it produces, to the switch. The switch can cache the content, which it receives from the host and, becomes a routing destination for that content on behalf of the host. This has to be communicated to the controller, for the controller to update its routing database with the content name and withthe route to retrieve that content. In our implementation, a switch sends out 'CONTENT_ANNOUCENMENT' message, whenever it caches a content and the controller updates its routing database with the necessary information extracted from the message. For the proof of concept, we have evaluated this scenario by sending out a content registration message from the host, when it gets attached to the switch and, the switch eventually sends out an announcement message to the controller. We have queried the controller routing table with the content name and ensured that the notification from the switch is updated in the routing database with the content name and the switch's identifier as the route for the content.

6.4.4 Proactive caching

'Proactive caching' is a process in which the controller pushes content into the caches of the switches. As explained in the implementation chapter, this can be used to ensure that popular contents are near to the user(for example, in high demand content broadcasting). We have proposed a cache in OF-ICN controller to store some of the popular contents. The controller pushes popular contents from it cache to the switches. In order to evaluate this behaviour, we have exploited the 'priority' field in controller cache. We have initialized the controller cache with a content with priority '1'. The proactive caching algorithm which runs in the controller periodically checks the priority of the cached contents in the controller. We ensured that the algorithm identifies this content with priority '1' in the cache and makes the controller to push the content to all the switches that are connected to it, for them to store the content in their cache. The switch's content store is verified before and after this process and, ensured that the priority content is stored by the switches.

6.4.5 Statistics

Our modified OpenFlow switch is enabled to keep track of the number of interest packets being processed for every content name stored in the FIB. The OpenFlow controller is expected to query for statistical data from the switch, for it to update its view on the underlying network. We have implemented the scenario wherein the controller will ask for the most used flow from the FIB of a switch. Based on the information returned by the switch, the controller will update its database and take further actions. For example, if the highly used flow has reached a benchmark packet count, the controller can push the content related to that flow, to the switch in order to reduce the round trip time. We have verified this behaviour with test case shown in Table 6.12.

6.5 Category 2 : Preserve OpenFlow functionalities

This section outlines the benchmarking points to ensure that the OpenFlow's basic working methodologies are preserved and additionally, they are used to enhance ICN communication

POX, being an OpenFlow platform, helps us to preserve the following OpenFlow functionalities in our implementation with some modifications :

1. **Switch-Controller handshake :** OpenFlow switches connect with the controller by swapping some information through messages like HELLO, FEATURES and CONFIG [6]. We are initializing the switch knowledge during this phase by bootstrapping the tables and dictionaries associated with the switch. This is verified by viewing the ICN related tables, after the initialisation phase and, ensuring that the necessary details are present.

Table 6.12: Evaluating statistics update from switch to controller

Test Scenario	Methodology	Result
Test that the controller can query for the most-used flow from the switch and update its local database. Also, test that controller can take caching decisions based on the statistical data	• **Topology :** This topology is same as that of Figure 4.1 . Initialize a flow_packet_count variable in controller with a benchmark value say, '50'. • Send out 'FIB_REQUEST' from controller • When the switch replies with the most used FIB entry,compare the packet count with 'flow_packet_count' variable's value • Ensure that if the value is greater than '50' controller pushes the related content to switch to store in its cache	Ensured that the controller can read the most used fib entry from a switch and compare it with the local variable. Ensured that if the value exceeds the benchmarked value, it can verify its own cache and if there a content cached for the corresponding flow, it can push the content to switch. Verified the values of switch's content store before and after executing this operation and ensured that it is updated with the content pushed from controller

2. **Capability advertisement :** During the switch connection with the controller, it will query for the features supported by the switch. The switch replies with few details; datapath id, supported number of tables and supported actions. Our implementation modified this information so that the switch can inform about some additional details like switch name and supported faces. This is verified by examining the corresponding dictionaries in controller after the handshake phase.

3. **Keep-alive with the controller :** Controller and switch connection is handled by a persistent IOWorker in POX. When a switch disconnects and reconnects again, the switch will automatically search for a controller and connect to it by utilising the available controller sockets. This functionality is ensured by killing the switch process multiple times and invoking it again

4. **Network functions separation :** We have maintained the main objective of Software-Defined Network platforms of keeping the controlling operations separate from the forwarding plane. In our implementation, the ICN-enabled OpenFlow switches carry out only processing and forwarding operations on the ICN packets based on the knowledge present in the ICN tables. All the controlling operations; forwarding decisions, caching decisions and FIB management are carried out by the controller. This is ensured by executing the controller test cases outlined in Appendix B.2.

5. **Flow based operations :** OpenFlow defines a flow, based on a rule, that contains a route for a packet type. It depends on a number of header fields in the packet. The same flow-level functionality is applied over ICN, by considering the 'content name' field in the ICN packets and framing the flows based on these content names, as they are the main primitives in ICN framework. Ensured that the ICN-enabled switch process the incoming ICN packets based on the flow rules present in the switch tables and drop the packets which do not match with any of the flow rules.

6. **Content announcement from the switch instead of a host :** As far as ICN is considered, the end nodes that connect to the network may have to announce the content directly to the network [29]. In our implementation, it is the switch that announces content availability to the controller. This can take advantage of aggregating many content announcements and the switch can send those in a single message, instead of overloading the controller with many messages. This is ensured by creating a list of contents names in the CONTENT_ANNOUNCEMENT message and send them altogether by the switch and, ensuring that the controller database is updated as expected

7. **Global view :** Unlike other ICN frameworks, ICN-enabled OpenFlow gives the advantage of providing the global view of the underlying network to a *'network monitoring application'* or a network operator. This global view is maintained by the control plane and used to make many controlling decisions including forwarding rule selection based on the network traffic, dropping a packet, caching decisions, and adaptive FIB management, unlike the existing ICN implementations, wherein the ICN daemon mostly takes forwarding decisions. This is verified by all the controlling test scenarios listed in the tables Table 7.13 and Appendix B.2 .

6.6 Category 3 : Implementation level verifications

Easy Deployment

Using the forked version of POX, we have modified necessary components to incorporate ICN functionalities over OpenFlow. The code for our implementation is available in GitHub repository as mentioned in section 5.10 . The deployment is a matter of cloning the GitHub repository and executing the commands as listed in that section. For easy evaluation, our implementation is targeted to run on a single machine. Thus all the components involved in our implementation; OpenFlow switch, OpenFlow controller and the protocol, all run in the same machine sharing the underlying system interfaces and configuration. Being an initiative study, this is done this way to give a better understanding and easy debugging. If the controller has to be run on a different machine, the switch command also has to be changed to look for a controller remotely. Due to the lack of Ethernet interfaces in a single machine, virtual interfaces are being used. If the switches need to be run on different machines, then the corresponding switch file has to be updated with the interfaces that are available in the machine in which it is running.

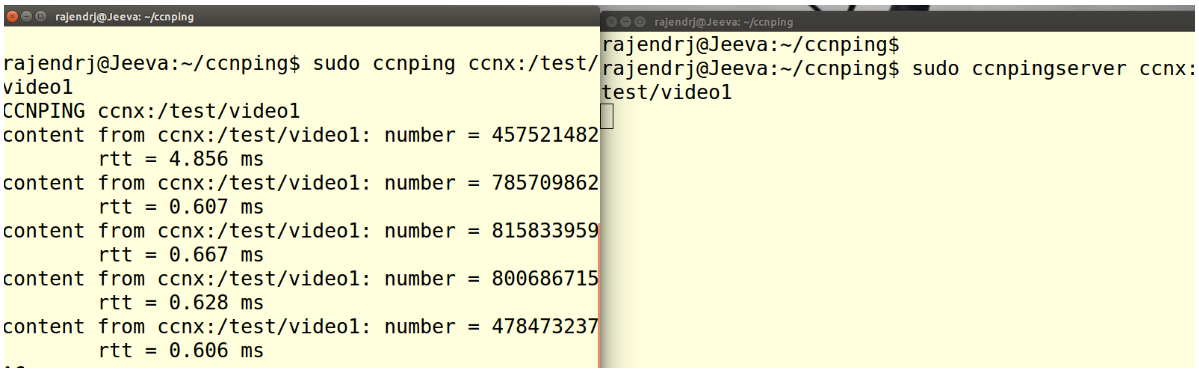
6.7 Category 4 : Compare with existing ICN implementations

This section evaluates our implementation against one of the most used ICN approach, CCN. We have used CCN's ping tool, 'CCNPing' and, CCN-based prototype, 'Mini-CCNx', to compare the functionalities of 'OF-ICN'

6.7.1 CCNx - CCNPing

CCNx provides a daemon *'ccnd'* which can be made running in a node to provide CCN functionalities. As part of the library, CCNx provides provisions for Face, FIB, PIT, Content Store and it supports generating interest and data packets. *'ccnping'* is a CCNx tool, impressed by tcp 'ping' application. This ccnping tool helps the experimenters to check the reachability of nodes in CCN environment through interest and data packets. We have evaluated our implementation against the functionalities provided by CCNx with respect to CCNPing tool and, the snapshots are provided here. CCNPing has a server module, that listens in a face for incoming interests for a particular 'named content' by adding the content name to the 'ccnd' daemon. The client module of CCNPing expresses interests for the named content produced by the server. The ccnd daemon will act upon the interest and hands it to the server. The server, in turn, sends back the data to the client through ccnd daemon. CCNPing application sends the pings in a loop until the process is killed. We have captured the *Round Trip Time (RTT)* , the pings take for every iteration and, compared it with the multiple interests sent out by the host implemented as part of our implementation. It is revealed that the time taken for the 1st ping iteration for a content is greater than the time taken for subsequent pings in both CCNx and our implementation. This is because of the utilisation of cache managed by ccnd daemon in CCNx and the cache in the switches, as part of our implementation.

Figures 6.8 and 6.9 show the comparison between CCNping and our implementation for multiple interests from a host:



```
rajendrj@Jeeva: ~/ccnping

rajendrj@Jeeva:~/ccnping$ sudo ccnping ccnx:/test/
video1
CCNPING ccnx:/test/video1
content from ccnx:/test/video1: number = 457521482
        rtt = 4.856 ms
content from ccnx:/test/video1: number = 785709862
        rtt = 0.607 ms
content from ccnx:/test/video1: number = 815833959
        rtt = 0.667 ms
content from ccnx:/test/video1: number = 800686715
        rtt = 0.628 ms
content from ccnx:/test/video1: number = 478473237
        rtt = 0.606 ms
```

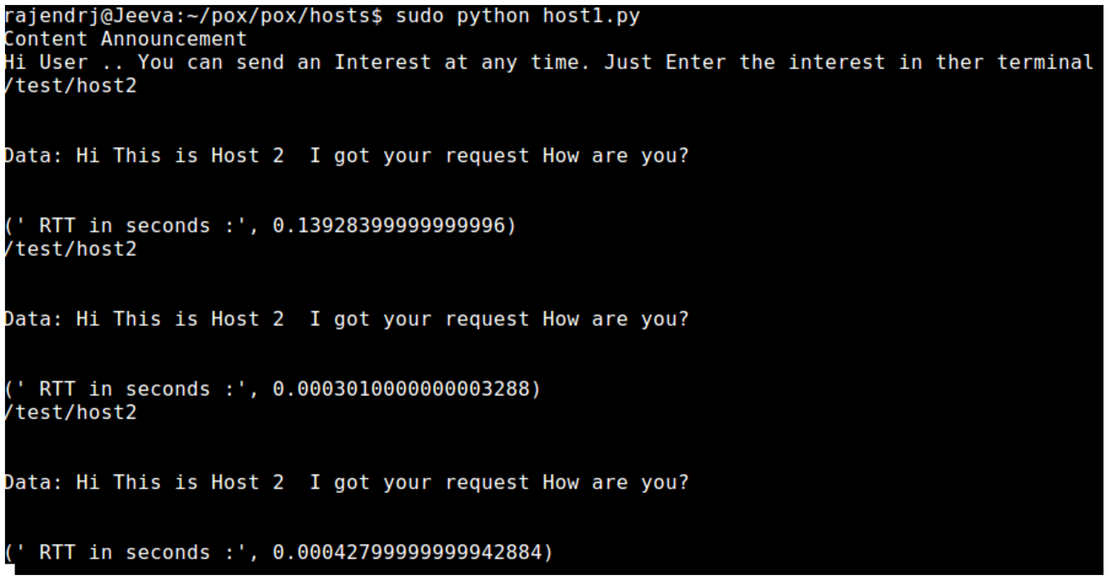
```
rajendrj@Jeeva: ~/ccnping
rajendrj@Jeeva:~/ccnping$
rajendrj@Jeeva:~/ccnping$ sudo ccnpingserver ccnx:
test/video1
```

Figure 6.8: OF-ICN and CCNPing comparison : Snapshot from CCNPing showing the RTT for multiple pings

Figure 6.8 shows that in CCNx, ccnping for the 2nd iteration takes lesser time(0.607 ms) than the 1st iteration(4.856 ms). Similarly, figure 6.9 shows the snapshot of Host 1 of our implementation, in which 2nd iteration takes less time (0.00030 seconds) compared to 1st iteration (0.13928 seconds) due to the utilisation of ICN cache in the modified OpenFlow switch. The values are plotted in the graph in Figure 6.10 .

6.7.2 Mini-CCNx

Mini-ccnx [43] is a mininet based ICN prototyping tool which is based on CCN architecture. Through CCN daemon, Mini-ccnx provides the basic architectural elements of ICN; FIB, PIT, and Content Store. It also maintains the advantages that come along



```
rajendrj@Jeeva:~/pox/pox/hosts$ sudo python host1.py
Content Announcement
Hi User .. You can send an Interest at any time. Just Enter the interest in ther terminal
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.13928399999999996)
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.00030100000000003288)
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.00042799999999942884)
```

Figure 6.9: OF-ICN and CCNPing comparison : Snapshot from OF-ICN showing the RTT for multiple interests from the same host

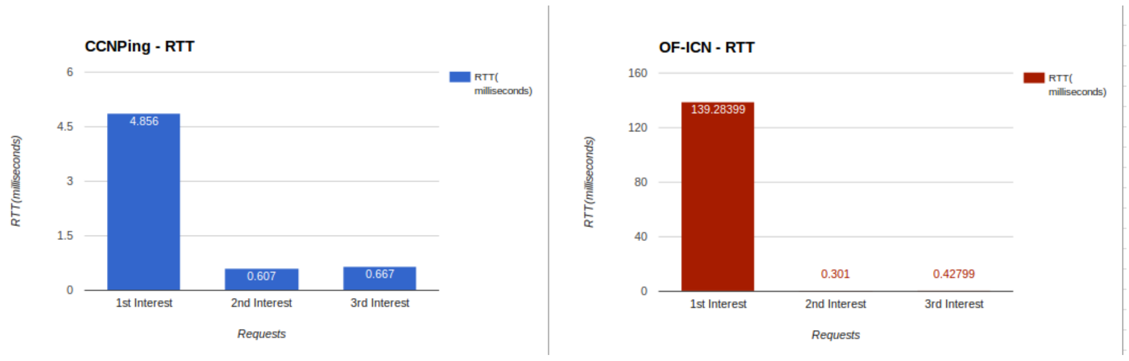


Figure 6.10: Graph comparing the RTT between CCNPing and OF-ICN for three subsequent requests

with Mininet such as the ability to run in a virtual environment, ability to define multiple topologies and link parameter modifications [56]. We have verified how well our implementation matches or differs from the ICN implementation experimented over Mini-ccnx.

We have created a topology using Mini-ccnx, as similar to our design architecture, as shown in figure 6.11.

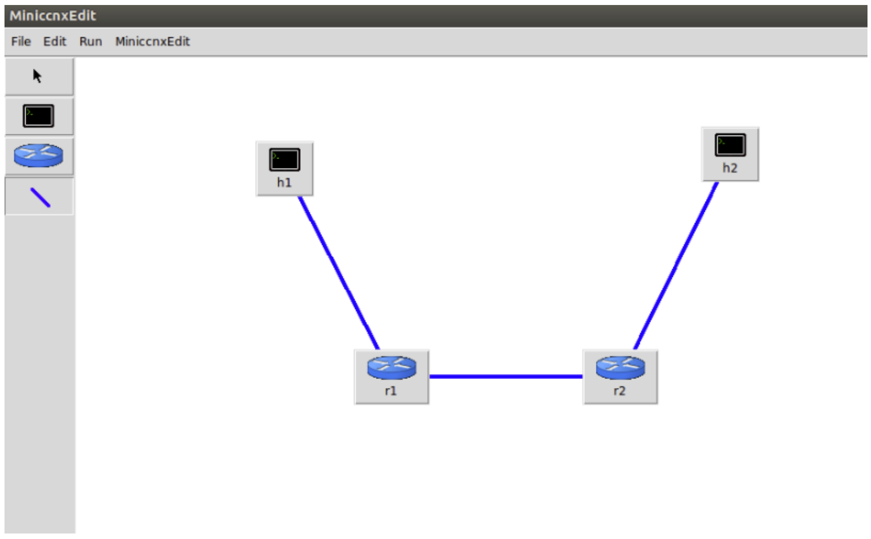
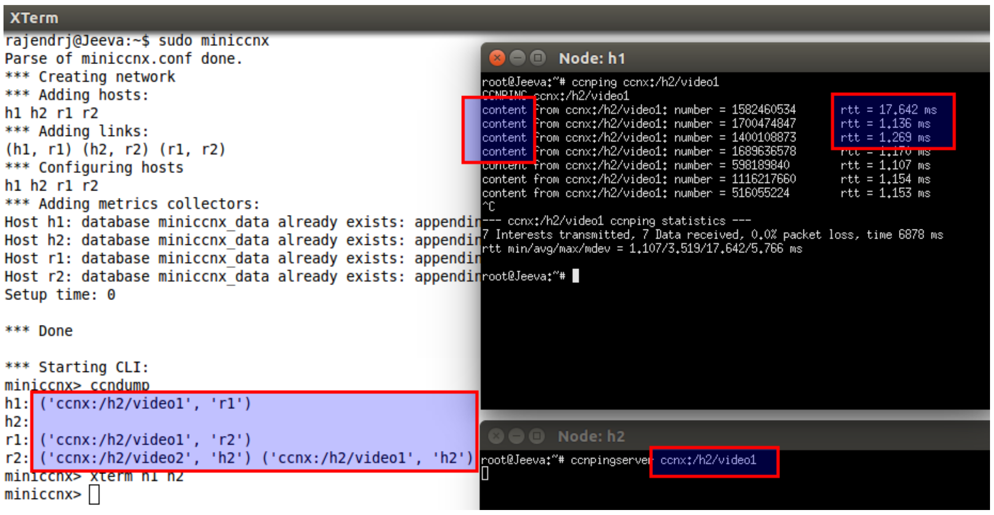


Figure 6.11: OF-ICN and Mini-CCNx comparison : Topology used for the experiment

Topology : Figure 6.11 shows that two switches r1 and r1 are connected to each other. Host h1 is connected to r1 and the host h2 is connected to r2. h1 produces two contents 'test/h2/video1' and '/test/h2/video2'. Both r1 and r2 have the forwarding details for '/test/h2/video1', while only r2 has the forwarding details for '/test/h2/video2'.

Test 1 : Host 1 sends out interests for the content '/test/h2/video1', for which both the switches have forwarding details

The behaviour which is showcased in the previous section holds for Test 1, which is that, the RTT takes for 1st interest for a content is greater than the subsequent requests in both Mini-ccnx and our solution. This behaviour shows that our implementation is able to provide the basic ICN functionalities which Mini-ccnx provides with CCNx. This behaviour is shown in Figure 6.12 and Figure 6.13, and the values are plotted in Figure 6.14 .

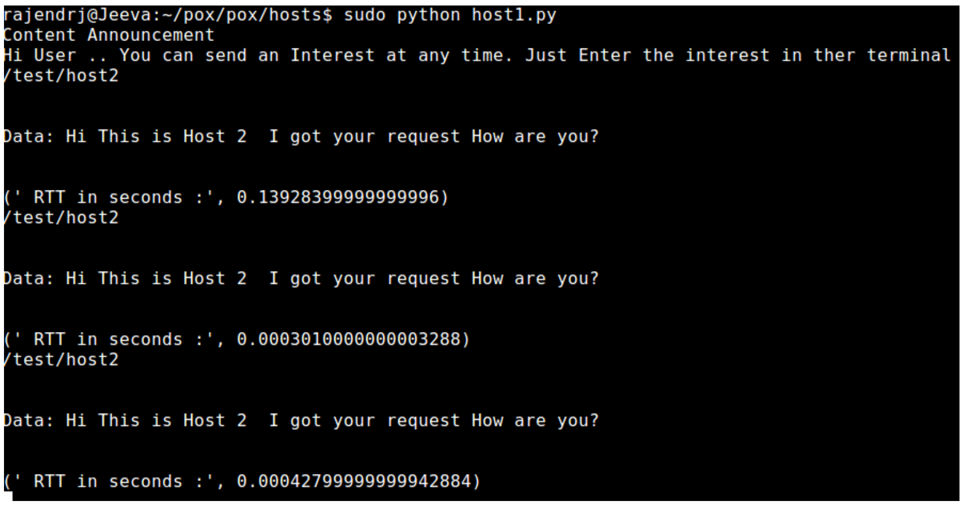


```
rajendrj@Jeeva:~$ sudo miniccnx
Parse of miniccnx.conf done.
*** Creating network
*** Adding hosts:
h1 h2 r1 r2
*** Adding links:
(h1, r1) (h2, r2) (r1, r2)
*** Configuring hosts
h1 h2 r1 r2
*** Adding metrics collectors:
Host h1: database miniccnx_data already exists: appendin
Host h2: database miniccnx_data already exists: appendin
Host r1: database miniccnx_data already exists: appendin
Host r2: database miniccnx_data already exists: appendin
Setup time: 0

*** Done

*** Starting CLI:
miniccnx> ccndump
h1: ('ccnx:/h2/video1', 'r1')
h2:
r1: ('ccnx:/h2/video1', 'r2')
r2: ('ccnx:/h2/video2', 'h2') ('ccnx:/h2/video1', 'h2')
miniccnx> xterm h1 h2
miniccnx>
```

```
root@Jeeva:~# ccnping ccnx:/h2/video1
CCNPING ccnx:/h2/video1
content from ccnx:/h2/video1: number = 1582460534     rtt = 17.642 ms
content from ccnx:/h2/video1: number = 1700474847     rtt = 1.136 ms
content from ccnx:/h2/video1: number = 1400108873     rtt = 1.269 ms
content from ccnx:/h2/video1: number = 1689636578     rtt = 1.170 ms
content from ccnx:/h2/video1: number = 598189840      rtt = 1.107 ms
content from ccnx:/h2/video1: number = 1116217660     rtt = 1.154 ms
content from ccnx:/h2/video1: number = 516055224      rtt = 1.153 ms
^C
--- ccnx:/h2/video1 ccnping statistics ---
7 Interests transmitted, 7 Data received, 0.0% packet loss, time 6878 ms
rtt min/avg/max/mdev = 1.107/3.519/17.642/5.766 ms

root@Jeeva:~#
```

```
root@Jeeva:~# ccnpingserver ccnx:/h2/video1
```

Figure 6.12: OF-ICN and Mini-CCNx comparison : Scenario where all intermediate switches have enough forwarding details for a content. Mini-CCNx receives content



```
rajendrj@Jeeva:~/pox/pox/hosts$ sudo python host1.py
Content Announcement
Hi User .. You can send an Interest at any time. Just Enter the interest in ther terminal
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.13928399999999996)
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.00030100000000003288)
/test/host2


Data: Hi This is Host 2  I got your request How are you?


(' RTT in seconds :', 0.00042799999999942884)
```

Figure 6.13: OF-ICN and Mini-CCNx comparison : Scenario where all intermediate switches have enough forwarding details for a content. OF-ICN host receives content

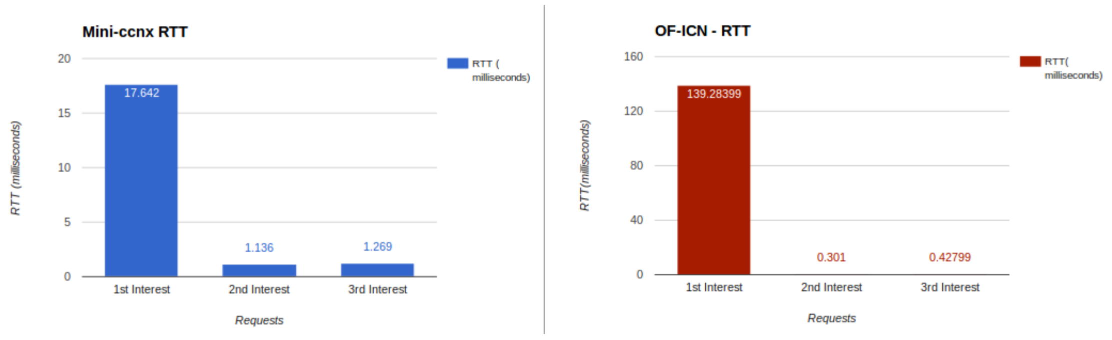
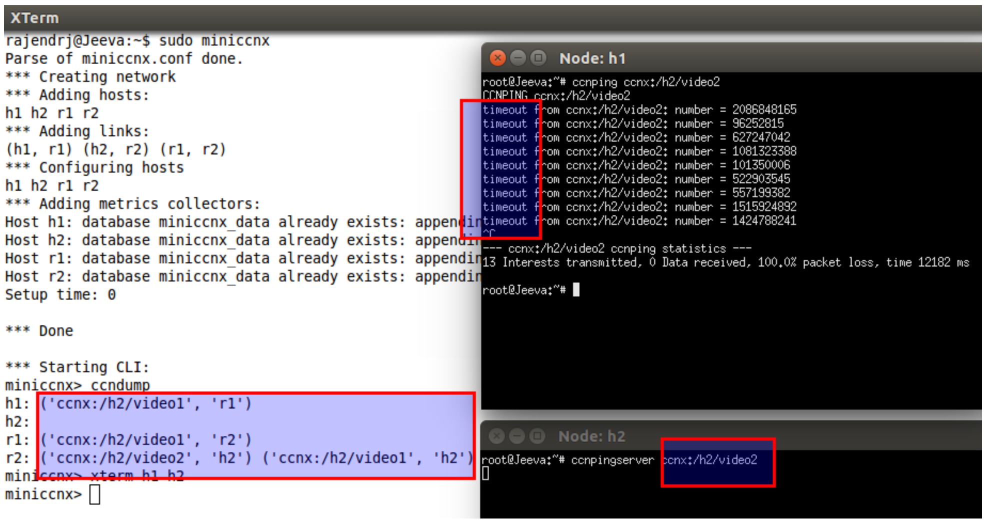


Figure 6.14: Graph comparing the RTT between Mini-CCNx and OF-ICN for three subsequent requests

Test 2 : In order to prove one obvious advantage of having a controller over other ICN implementations, we have verified a scenario in which for a content ’C’, which h2 provides, only the switch r2 has the forwarding route and Switch r1 is agnostic to the route. Now, h1 is sending out an interest for the content ’C’ to switch r1. We have created the above scenario in both Mini-ccnx and our solution ’OF-ICN’ and compared how they reacted to this scenario. The results show that Mini-ccnx ’times out’ for the interests as the ccnd daemon that runs in switch r1 is not able to identify the forwarding route for the interest. This is because r1 does not have an entry for the content in its FIB. Until otherwise a routing protocol again runs and updates the FIBs in all the switches, Mini-ccnx will not be able to identify where to send the interest, for which one of the other connected switches in the path has the route. This is shown in Figure 6.15 .

On the other hand, our modified OpenFlow can successfully deliver the interest to the host H2. Controller’s routing database comes as a solution here. This is because when r2 receives the content announcement from H2 for ’C’, it updates its FIB and informs the controller as well. In the other way, controller frequently queries for features from switches and, updates its routing database. Thus, when r1 finds out that it does not have a matching entry for the interest from H1, it sends out a query to the controller. The controller, by scanning its routing database, learns that r2 is the route for the content and updates switch r1 accordingly. Now, the switch r1 will be able to send the interest



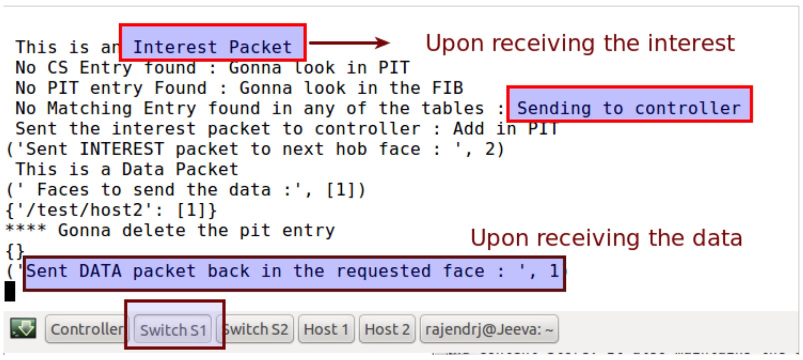
```
XTerm
rajendrj@Jeeva:~$ sudo miniccnx
Parse of miniccnx.conf done.
*** Creating network
*** Adding hosts:
h1 h2 r1 r2
*** Adding links:
(h1, r1) (h2, r2) (r1, r2)
*** Configuring hosts
h1 h2 r1 r2
*** Adding metrics collectors:
Host h1: database miniccnx_data already exists: appendir
Host h2: database miniccnx_data already exists: appendi
Host r1: database miniccnx_data already exists: appendir
Host r2: database miniccnx_data already exists: appendir
Setup time: 0

*** Done

*** Starting CLI:
miniccnx> ccndump
h1: ('ccnx:/h2/video1', 'r1')
h2:
r1: ('ccnx:/h2/video1', 'r2')
r2: ('ccnx:/h2/video2', 'h2') ('ccnx:/h2/video1', 'h2')
miniccnx> xterm h1 h2
miniccnx>
```

```
Node: h1
root@Jeeva:~# ccnping ccnx:/h2/video2
CCNPING ccnx:/h2/video2
timeout from ccnx:/h2/video2: number = 2086848165
timeout from ccnx:/h2/video2: number = 96252815
timeout from ccnx:/h2/video2: number = 627247042
timeout from ccnx:/h2/video2: number = 1081323388
timeout from ccnx:/h2/video2: number = 101350006
timeout from ccnx:/h2/video2: number = 522903545
timeout from ccnx:/h2/video2: number = 557199382
timeout from ccnx:/h2/video2: number = 1515924892
timeout from ccnx:/h2/video2: number = 1424788241
^C
--- ccnx:/h2/video2 ccnping statistics ---
13 Interests transmitted, 0 Data received, 100.0% packet loss, time 12182 ms

root@Jeeva:~#
```

```
Node: h2
root@Jeeva:~# ccnpingserver ccnx:/h2/video2
```

Figure 6.15: OF-ICN and Mini-CCNx comparison : Scenario where some intermediate switches have less forwarding details. Mini-CCNx times out

to H2 through r2. This is shown in Figure 6.16 .



```
 This is an Interest Packet
 No CS Entry found : Gonna look in PIT
 No PIT entry Found : Gonna look in the FIB
 No Matching Entry found in any of the tables : Sending to controller
 Sent the interest packet to controller : Add in PIT
('Sent INTEREST packet to next hob face : ', 2)
 This is a Data Packet
(' Faces to send the data :', [1])
{'/test/host2': [1]}
**** Gonna delete the pit entry
{}
('Sent DATA packet back in the requested face : ', 1
```

Figure 6.16: OF-ICN and Mini-CCNx comparison : Scenario where some intermediate switches have less forwarding details. OF-ICN host receives content

6.8 Summary

This chapter portrayed the evaluations carried out to verify the proposed OF-ICN approach. We have divided the evaluation process into four categories :

1. To ensure that OF-ICN provides the essential ICN functionalities

2. To ensure that the successful OpenFlow features are preserved in our implementation and they are used to enhance the ICN functionalities

3. To ensure that the implementation is easy to deploy and debug

4. To compare and evaluate OF-ICN with existing ICN prototypes

Under each category, various test scenarios are considered and documented. The test cases are evaluated with OF-ICN and the results are depicted in this chapter. A summary of the test cases which are utilised for evaluating our implementation is shown in Table 6.13. Each test case states the test scenario, the result and the OpenFlow messages and, events or actions that are involved as part of the test case. The test cases that are specific to the OpenFlow switches and the OpenFlow controller are covered in Appendix B. The results show that, with the necessary extensions to the OpenFlow elements (OpenFlow switch, OpenFlow controller and OpenFlow protocol), OF-ICN could successfully provide an OpenFlow-based control plane for the underlying ICN-enabled OpenFlow switches. It is shown that, OF-ICN could provide the necessary ICN functionalities, according to the generic ICN features derived by us, by considering different ICN implementations. This chapters also depicted that, the successful OpenFlow features are still preserved in OF-ICN and modified in some places to add ICN-related information to them. The outcomes out of the comparison between OF-ICN and a number of ICN prototypes reassured that, OF-ICN could provide enough essential ICN capabilities and, with the presence of the centralised controller, OF-ICN could show several advantages over the existing ICN prototypes.

Table 6.13: Functional Test Cases used to evaluate OF-ICN features

S.No	**Test Case**	**Sub Test Case**	**Action**	**Message/ Event/ Action**
1	Content Store Match		Send Data packet back in incoming face	PACKET_OUT[FACE]
2	PIT Match		Add the incoming face to waiting list	ADD_PIT[FACE]
3	FIB Match		Forward the interest in outgoing face	PACKET_OUT[FACE]
4	FIB Match		Add the incoming face to PIT	ADD_PIT[FACE]
5	No match in switch	Controller has cached data	Send the data packet from controller to switch	OFPT_DATA_FROM _CONTROLLER
6	No match in switch	Controller has the route	Send Flow Rule to switch	FLOW_MOD[RULE] PACKET_OUT[FACE]
7	Second interest for the same content		Match with FIB in the switch and send the packet in outgoing face	PACKET_OUT[FACE]
8	Data packet	Matches with PIT	Send the data packet back in the waiting face	PACKET_OUT[FACE]
9	Data packet	No match with PIT	Drop the data packet	
10	Content Store Full		Message from controller to clear the cache in switch	OPFT_CS_FULL OFPT_CLEAR_CS
11	Proactive caching		Cache the content from controller in the switch	OFPT_ADD_CS_ENTRY
12	Content Announcement		Store the content name and the switch identifier in controller routing database	OFPT_CONTENT _ANNOUNCEMENT

Chapter 7

Conclusions

'OF-ICN' is the experimental solution proposed by this dissertation to provide an OpenFlow-based control plane for Information-Centric Networking(ICN) platform, in order to realise a clean state integration between OpenFlow and ICN. This study demonstrated this by porting the basic ICN functionalities over OpenFlow architecture and showed that the ICN-enabled OpenFlow controller and the ICN-enabled OpenFlow switch, can successfully handle ICN communication in terms of flows, using the ICN-enabled OpenFlow protocol. This is achieved by modifying the following major elements provided by OpenFlow:

- OpenFlow switch
- OpenFlow controller
- OpenFlow protocol

'OF-ICN' is implemented and evaluated using POX, a Python OpenFlow controller. It used the switch, controller and the protocol library component modules provided by POX and, extended them to realise ICN-based functionalities. As part of the evaluation, 'OF-ICN' is compared with a number of ICN-based prototypes to ensure that the ICN-enabled OpenFlow actually provides the core ICN features and moreover, to ensure that the modified OpenFlow can show benefits over existing ICN implementations in a number of scenarios. The results showed that, by necessary modifications to the OpenFlow elements, the proposed approach OF-ICN, could successfully provide an OpenFlow-based control plane for ICN-enabled OpenFlow switches. OF-ICN could make forwarding and caching decisions for the underlying ICN-enabled datapath infrastructure. The comparison with existing ICN prototypes revealed that, an ICN implementation with a control plane such as OpenFlow, can show improved packet processing and delivery, compared to the existing ICN prototypes.

This chapter portrays the contributions of this dissertation towards providing a solution for the integration between OpenFlow and ICN by creating an OpenFlow-based control plane, followed by the limitations of the study and the future research directions. Finally, the chapter ends with denoting some of the long term goals for this study.

7.1 Contributions

This section outlines the major contributions of this dissertation towards achieving the goal of creating an OpenFlow-based control plane for ICN :

- **Analysis of gaps**

  This dissertation has clearly studied the existing works in literature towards realising a clean state integration between two emerging technologies; OpenFlow and ICN. The background, various implementations and the tools behind these technologies are studied as part of this dissertation. In order to give a better formulation, this study analysed the works that provided a quick realisation of the integration, by providing workarounds and by not modifying the architecture of OpenFlow and ICN. The drawbacks of these non-extension approaches are identified and taken as the root motivation for the works on extension-based approaches. The extension-based approaches tried to modify OpenFlow to incorporate ICN features. The pros and cons of these extension-based approaches are analysed and, the gaps are identified and listed as part of this dissertation.

- **Formulate generic ICN features over OpenFlow**

  From the gaps identified through the analysis of existing works, this dissertation segregated the generic ICN-based functionalities and proposed 'OF-ICN': a solution to fill these gaps, by modifying the existing OpenFlow architecture, to accommodate the identified generic ICN functionalities. A clean state integration approach is suggested by this study, instead of relying on workarounds, wrappers or plugins, which could only provide a short-term overlay realisation of the required functionalities.

- **Extensions and algorithms**

  This dissertation presented the possible extensions to the architecture of the major OpenFlow elements: OpenFlow switch and OpenFlow controller, to enable them to support ICN forwarding and caching operations. This is achieved by equipping them with a new set of data structures to store ICN-related knowledge. The algorithms to make forwarding decisions for ICN flows as part of the controller, and to execute forwarding operations for ICN flows as part of the switch, are represented in this dissertation.

- **Extensions to OpenFlow protocol**

  Having presented the modifications to the OpenFlow switch and the controller, the dissertation moved on to list the new programming interface components needed to communicate the ICN-related information between the switch and the controller. This enabled the proposed approach to have a control layer based on OpenFlow, to instruct the datapath layer of ICN-enabled OpenFlow switches, and to send and receive ICN-based details from the underlying network.

- **Modularized implementation**

  To realise the aim of providing a modularized implementation, POX, a software OpenFlow controller in Python language, is selected by this study, to experiment and evaluate the proposed approach. This dissertation exploited the software components provided by POX to realise various OpenFlow elements; switch, controller and the protocol library. All the proposed modifications and extensions on OpenFlow by this dissertation, are experimented over these components and the modularization is ensured between the components, to enable them to be plugged in into any future experiments around this study.

7.2 Future works

This section outlines the limitations of this study and future research directions to overcome those limitations and to provide enhancements:

- **Supporting a standardised ICN naming scheme**

  As outlined in the chapter on literature, the naming scheme for ICN is still an active research area and there is no global consensus on the naming methodology to be used to realise a global ICN architecture [29, 44]. Given that, this has been a challenge to almost all the existing works in the literature on ICN and to the existing works on integration between OpenFlow and ICN. This challenge applies to this study as well. We have used Ethernet packets to carry ICN-related information and enabled the software switches to do deep packet inspection to traverse the ICN-related information in the packet. We are aware that this will be difficult to convert into a hardware implementation of the switch, as the current hardware switches lag deep packet inspection property. We are denoting this naming part as a future research opportunity over the proposed approach and it can be fully realised when ICN naming scheme is standardised.

- **Performance tuning**

  Python, as an easy-to-code language, is not optimised well enough to run the code faster. POX is built completely on Python, which results in slow speed code processing in POX. NOX [61] is a counterpart OpenFlow controller to POX, which is developed in C++, from which POX is derived. Even though they provide similar functionalities, NOX runs better than POX controller in terms of code-level executions. There are some libraries and tools which are available to improve the performance of Python (For example, PyPy library [105], Numba [106]). These libraries and tools can be used over POX, to improve the performance, which in turn will provide better processing speed for our approach as well.

- **Security and integrity**

  ICN claims to provide inbuilt integrity and security within each content in the network, by incorporating digital signature and signature-related information in ICN packets, and by encrypting the packets. The network elements and the consumers can utilise these options, to ensure that the content is the intended content and, it is from the legitimate user. Limiting our dissertation scope to experiment basic ICN features, we have not provided these options in our implementation, other than the binary conversion of OpenFlow messages. These security and integrity features can be built over our approach, to providing the assurance over the contents being communicated in the network.

- **Scalability**

  SDN and its approach, OpenFlow, suggest a centralised controller to control the underlying network of switches. When we equip the network with highly capable ICN-enabled switches, a number of details and actions which the controller has to handle will increase compared to the traditional IP-based OpenFlow controller. Thus, there seems to be a need to provide a distributed control layer with a number of controllers connected, to provide a scalable controlling layer over the underlying network elements. Few research activities in literature tried to providing such scalable control layer which can also be applied to our approach to scaling it better for future requirements [36]

7.3 Final remarks

SDN and OpenFlow, are designed to support the current needs of network operators, developers and researchers. Whereas, ICN gives the awareness to the Internet community, about the need for an architectural change in the underlying inter-network and it claims to support the future needs of the Internet users. It is very difficult to deny completely any of the claims made by both OpenFlow and ICN technologies, as there are a number of studies and research activities, that prove that those claims are true in some scenario or the other. The 'future', which ICN has mentioned, has already arrived and thus, there is a lot of attention from industry and academia to realise a network that scales well with the surge of enormous data and is flexible to incorporate new control and traffic engineering policies. OpenFlow and ICN approaches are picked in this ground and being actively investigated to port one over the other, to realise a secure network with more flexibility, availability and high manageability. We believe that this dissertation work, with its experimental approach and solution, will be a good contribution towards this goal of integrating both the technologies. We also believe that this study will give a good starting point to any future research on this platform. Keeping that in mind, a long-term goal of this study is to provide an addendum document to the latest OpenFlow specification, by clearly marking the requirements of the OpenFlow switch specification, to support ICN packet switching. This document should cover the extended components of OpenFlow switches and the OpenFlow protocol changes, that are needed to control those switches from a controller, which could be local or remote.

Appendix A

Abbreviations

Short Term	Expanded Term
ICN	Information-Centric Networking
SDN	Software Defined Networking
FIB	Forwarding Information Base
PIT	Pending Interest Table
CS	Content Store
OF	OpenFlow
IP	Internet Protocol
NDN	Named Data Networking
CCN	Content Centric Networking

Appendix B

Switch and Controller use cases

B.1 Switch use cases

B.2 Controller use cases

Table B.1: Switch use cases

S.No	Test	Matching table	Expected Result
1	Interest packet received	Content Store Match	Send the data back to requester
2	Interest packet received	PIT Match	Add the incoming face(interface) to the list of waiting faces
3	Interest packet received	FIB Match	Forward the interest to the list of next hop faces
4	Interest packet received	No Match	Send the interest to controller
5	Data packet received	Content Store Match	Duplicate content - Do not add to the cache
6	Data packet received	PIT Match	Send the data out in the listed waiting faces
7	Data packet received	No PIT Match	Drop the packet
8	Content Store Full		Send 'CS_FULL' message to controller
9	Received 'ADD_PIT' message from controller		Add the entry into PIT table
10	Received 'ADD_CS_ENTRY' message from controller		Add the entry into Content Store cache
11	Received 'CLEAR_CS' message from controller		Delete all the entries from Content Store
11	Data packet received from controller		Send the packet out in the waiting faces and cache the content

Table B.2: Controller use cases

S.No	**Test**	**Expected Result**
1	Received Interest packet from switch	Check for the content in controller cache. If no cache, check for an entry in routing database
2	Entry found in controller cache for an interest	Send the data back to the switch and push a FLOW_MOD message
3	Entry found in routing database	Send FLOW_MOD and PACKET_OUT messages to switch
4	No information found for the interest	Ask the switch to drop the packet
5	CS_FULL message received from switch	Send CLEAR_CS message [an algorithm to select the entries to be deleted]
6	Data packet received	Send the data out in the listed waiting faces
7	Received content announcement from a switch	Update the routing database

Bibliography

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[2] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, S. Shenker, and I. U. C. Berkeley, "Software-defined internet architecture: decoupling architecture from infrastructure," *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 43–48, 2012.

[3] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, and S. Louis, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.

[4] V. Jacobson, D. K. Smetters, N. H. Briggs, J. D. Thornton, M. F. Plass, and R. L. Braynard, "Networking Named Content," *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies CoNEXT '09*, pp. 1–12, 2009.

[5] 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.

[6] OpenFlow1.0.0, "OpenFlow Switch Specification, Version 1.0.0(Wire Protocol 0x01)," *http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf*, 2009.

[7] OpenFlow1.2.0, "OpenFlow Switch Specification, Version 1.2 (Wire Protocol 0x03)," *Open Networking Foundation, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf*, 2011.

[8] OpenFlow1.3.0, "OpenFlow Switch Specification, Version 1.3.0(Wire Protocol 0x04)," *Open Networking Foundation, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf*, 2012.

[9] OpenFlow1.4.0, "OpenFlow Switch Specification, Version 1.4.0(Wire Protocol 0x05)," *Open Networking Foundation, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf*, 2013.

[10] X. N. Nguyen, D. Saucez, and T. Turletti, "Efficient caching in content-centric networks using OpenFlow," *Proceedings - IEEE INFOCOM*, pp. 1–2, 2013.

[11] N. L. M. Van Adrichem and F. A. Kuipers, "NDNFlow: Software-defined named data networking," *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT 2015*, 2015.

[12] 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.

[13] J. Suh, "OF-CCN : CCN over OpenFlow," *NDN hands-on Workshop*, 2012.

[14] A. Chanda and C. Westphal, "Content Based Traffic Engineering in Software Defined Information Centric Networks," *INFOCOM, 2013. 32nd IEEE International Conference on Computer Communications*, pp. 3397–3402, 2013.

[15] J. Mccauley, "Pox: A python-based openflow controller," *https://github.com/noxrepo/pox*, 2014.

[16] T.-y. Feng, "A survey of interconnection networks," *Computer*, vol. 14, pp. 12—-27, 1981.

[17] R. Froom, B. Sivasubramanian, and E. Frahim, *Implementing Cisco IP Switched Networks (SWITCH) Foundation Learning Guide: Foundation Learning for SWITCH 642-813.* Cisco press, 2010.

[18] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "Simple network management protocol (SNMP)," *No. RFC 1157*, 1990.

[19] J. Moy, "OSPF version 2," 1997.

[20] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," 2005.

[21] B. A. A. Nunes, M. Mendonca, X.-n. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking : Past , Present , and Future of Programmable Networks," pp. 1–18, 2014.

[22] H. Farhady, H. Lee, and A. Nakao, "Software-Defined Networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[23] R. Ravindran and G.-q. Wang, "Software-Defined Information Centric Network (ICN)," 2016.

[24] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[25] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in openflow networks," *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 177—-188, 2014.

[26] F. Hu, Q. Hao, and K. Bao, "A Survey on Software Defined Networking (SDN) and OpenFlow: From Concept to Implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. c, pp. 1–1, 2014.

[27] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible OpenFlow-controller benchmark," *2012 European Workshop on Software Defined Networking*, pp. 48—-53, 2012.

[28] V. Jacobson, "A new way to look at networking," *Google Tech Talk*, vol. 30, 2006.

[29] D. Saxena, V. Raychoudhury, N. Suri, C. Becker, and J. Cao, "Named Data Networking: A survey," *Computer Science Review*, vol. 19, pp. 15–55, 2016.

[30] C. E. Perkins, "Mobile networking through mobile IP," *IEEE Internet Computing*, vol. 2, no. 1, pp. 58–69, 1998.

[31] V. Jacobson, J. Burke, L. Zhang, K. Claffy, C. Papadopoulos, L. Wang, J. A. Halderman, and P. Crowley, "Named Data Networking Next Phase (NDN-NP) Project May 2014 - April 2015 Annual Report," no. April, 2015.

[32] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communications Surveys \& Tutorials*, vol. 16, pp. 1024—-1049, 2014.

[33] D. Barkai, "An introduction to peer-to-peer computing," *Intel Developer update magazine*, pp. 1—-7, 2000.

[34] K. Dirk, F. Hannu, and K. Holger, "Information-Centric Networking A position paper ," pp. 1–2, 2010.

[35] M. Tortelli, D. Rossi, G. Boggia, and L. A. Grieco, "ICN software tools: Survey and cross-comparison," *Simulation Modelling Practice and Theory*, vol. 63, pp. 23–46, 2016.

[36] A. El Mougy, “On the Integration of Software-Defined and Information-Centric Networking Paradigms,” *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 105–110, 2015.

[37] N. Melazzi, a. Detti, G. Mazza, G. Morabito, S. Salsano, and L. Veltri, “An OpenFlow-based testbed for information centric networking,” *Future Network & Mobile Summit (FutureNetw), 2012*, pp. 1–9, 2012.

[38] D. Syrivelis, G. Parisis, D. Trossen, P. Flegkas, V. Sourlas, T. Korakis, and L. Tassiulas, “Pursuing a software defined information-centric network,” *Proceedings - European Workshop on Software Defined Networks, EWSDN 2012*, pp. 103–108, 2012.

[39] M. Arumaithurai, J. Chen, E. Monticelli, X. Fu, and K. K. Ramakrishnan, “Exploiting ICN for flexible management of software-defined networks,” *Proceedings of the 1st international conference on Information-centric networking - INC ’14*, pp. 107–116, 2014.

[40] H. Luo, J. Cui, Z. Chen, M. Jin, and H. Zhang, “Efficient integration of software defined networking and information-centric networking with CoLoR,” *2014 IEEE Global Communications Conference, GLOBECOM 2014*, pp. 1962–1967, 2015.

[41] D. Perino and M. Varvello, “A reality check for content centric networking,” *Proceedings of the ACM SIGCOMM workshop on Information-centric networking - ICN ’11*, p. 44, 2011.

[42] Ccnping, “ccnping,” *University Of Arizona, https://github.com/NDN-Routing/ccnping.*

[43] C. M. S. Cabral, C. E. Rothenberg, and M. F. Magalhães, “Mini-{CCNx:} fast prototyping for named data networking,” *Proceedings of the 3rd {ACM} {SIGCOMM} workshop on Information-centric networking*, pp. 33–34, 2013.

[44] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, L. Wang, P. Crowley, and E. Yeh, "Named Data Networking (NDN) Project," no. October, 2010.

[45] M. Mosko, "CCNx 1 . 0 Protocol Specifications Roadmap," vol. 2013, no. rev 2, pp. 1–9, 2013.

[46] A. Detti, N. Blefari-Melazzi, S. Salsano, and M. Pomposini, "CONET: A Content Centric Inter-Networking Architecture," *Proceedings of the ACM SIGCOMM workshop on Information-centric networking - ICN '11*, pp. 50–55, 2011.

[47] J. Postel, "Internet protocol," 1981.

[48] A. Afanasyev, J. Burke, L. Zhang, K. Claffy, L. Wang, V. Jacobson, P. Crowley, C. Papadopoulos, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 66—-73, 2014.

[49] A. Afanasyev, Y. Yu, L. Zhang, J. Burke, kc Claffy, and J. Polterock, "The Second Named Data Networking Community Meeting (NDNcomm 2015)," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 58–63, 2016.

[50] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "NLSR: Named-data Link State Routing Protocol," *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking - ICN '13*, p. 15, 2013.

[51] L. Wang, A. Hoque, C. Yi, A. Alyyan, and B. Zhang, "OSPFN: An OSPF based routing protocol for Named Data Networking," *University of Memphis and University of Arizona, Tech. Rep*, 2012.

[52] S. Skiena, "Dijkstra's algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225—-227, 1990.

[53] M. F. Bari, R. Chowdhury, Shihabur Rahman Ahmed, R. Boutaba, and B. Mathieu, "A survey of naming and routing in information-centric networks," *IEEE Communications Magazine*, vol. 50, pp. 44—-53, 2012.

[54] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," *Workshop on Hot Topics in Networks*, pp. 1–6, 2010.

[55] D. Byun, B.-J. B. Lee, and M.-W. Jang, "Adaptive flow control via Interest aggregation in CCN," *2013 IEEE International Conference on Communications (ICC)*, pp. 3738—-3742, 2013.

[56] C. Cabral, C. E. Rothenberg, and M. F. Magalhaes, "High fidelity content-centric experiments with Mini-CCNx," *Proceedings - International Symposium on Computers and Communications*, 2014.

[57] A. Gawande, V. Lehman, Y. Zhang, L. Wang, J. Shi, B. Zhang, and A. Afanasyev, "Mini-NDN GitHub," 2015.

[58] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, A. Networks, and M. Casado, "The Design and Implementation of Open vSwitch," *12th USENIX Symposium of Networked Systems Design and Implementation*, pp. 117–130, 2015.

[59] E. L. Fernandes, "OpenFlow 1.3 software switch," *https://github.com/CPqD/ofsoftswitch13*, 2012.

[60] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers," no. JANUARY 2014, 2015.

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

[62] S. Kaur, J. Singh, and N. S. Ghumman, "Network Programmability Using POX Controller," *International Conference on Communication, Computing & Systems*, p. 5, 2014.

[63] Ryu, "Ryu," *https://osrg.github.io/ryu/.*

[64] Floodlight, "FloodLight," *http://www.projectfloodlight.org/floodlight/.*

[65] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014.

[66] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using Mininet for emulation and prototyping Software-Defined Networks," *2014 IEEE Colombian Conference on Communications and Computing, COLCOM 2014 - Conference Proceedings*, 2014.

[67] OpenFlow1.5.1, "OpenFlow specification 1.5.1 (Protocol version 0x06)," *Open Networking Foundation*, 2014.

[68] M. Vahlenkamp, F. Schneider, D. Kutscher, and J. Seedorf, "Enabling Information-Centric Networking in IP Networks Using SDN," *Future Networks and Services (SDN4FNS), IEEE*, pp. 1–6, 2013.

[69] L. Veltri, G. Morabito, S. Salsano, N. Blefari-Melazzi, and A. Detti, "Supporting information-centric functionality in software defined networks," *IEEE International Conference on Communications*, pp. 6645–6650, 2012.

[70] S. Das, "Extensions to the OpenFlow protocol in support of circuit switching," *OpenFlow protocol specification (v1. 0)Circuit Switch*, 2010.

[71] J. Wang, J. Ren, K. Lu, J. Wang, S. Liu, and C. Westphal, "An optimal Cache management framework for information-centric networks with network coding," *2014 IFIP Networking Conference*, no. 61202378, pp. 1–9, 2014.

[72] M. Shen, B. Chen, X. Zhu, and Y. Zhao, "Towards Optimal Cache Decision for Campus Networks with Content-Centric Network Routers," *2016 IEEE Symposium on Computers and Communication (ISCC)*, pp. 810–815, 2016.

[73] A. Köpsel and H. Woesner, "OFELIA - Pan-European test facility for OpenFlow experimentation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (W. Abramowicz, I. M. Llorente, M. Surridge, A. Zisman, and J. Vayssière, eds.), vol. 6994 LNCS, pp. 311–312, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[74] M. Sune, L. Bergesio, H. Woesner, T. Rothe, A. Kopsel, D. Colle, B. Puype, D. Simeonidou, R. Nejabati, M. Channegowda, M. Kind, T. Dietz, A. Autenrieth, V. Kotronis, E. Salvadori, S. Salsano, M. Korner, and S. Sharma, "Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed," *Computer Networks*, vol. 61, pp. 132–150, 2014.

[75] S. Salsano, N. Blefari-melazzi, V. Luca, B. Stefano, A. Araldo, F. Patriarca, M. Bonola, and S. Signorello, "Ofelia Deliverable 9.1 EXOTIC final architecture and design," no. 43, pp. 1–43, 2012.

[76] ALIEN, "Hardware Abstraction Layer," vol. 2015, no. February 11, 2007.

[77] E. Jacob, V. Fuentes, J. Matías, and M. H, "CCN and ALIEN developments integration over OFELIA," tech. rep., 2012.

[78] X. N. Nguyen, D. Saucez, T. Turletti, X. N. Nguyen, D. Saucez, T. Turletti, and P. Ccn, "Providing CCN functionalities over OpenFlow switches," 2013.

[79] S. Eum, M. Jibiki, M. Murata, H. Asaeda, and N. Nishinaga, "A design of an ICN architecture within the framework of SDN," *International Conference on Ubiquitous and Future Networks, ICUFN*, vol. 2015-Augus, pp. 141–146, 2015.

[80] A. Ooka, S. Ata, T. Koide, and H. Shimonishi, "OpenFlow-based Content-Centric Networking Architecture and Router Implementation," *Future Network and Mobile Summit (FutureNetworkSummit), 2013*, pp. 1–10, 2013.

[81] S. Charpinel, C. Alberto, S. Santos, M. Martinello, and R. Villaca, "SDCCN : A Novel Software Defined Content-Centric Networking Approach," *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 87–94, 2016.

[82] H. Song, "Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane," *HotSDN '13*, pp. 127–132, 2013.

[83] R. Ravindran, X. Liu, A. Chakraborti, X. Zhang, and G. Wang, "Towards software defined ICN based edge-cloud services," *Proceedings of the 2013 IEEE 2nd International Conference on Cloud Networking, CloudNet 2013*, pp. 227–235, 2013.

[84] H. Luo, Z. Chen, J. Cui, H. Zhang, and M. Jin, "Color: an information-centric internet architecture for innovations," *IEEE Network*, vol. 3, no. June, pp. 4–10, 2014.

[85] S. Shailendra, B. Panigrahi, H. K. Rath, and A. Simha, "A novel overlay architecture for Information Centric Networking," *2015 21st National Conference on Communications, NCC 2015*, 2015.

[86] F. Bacher, B. Rainer, and H. Hellwagner, "Towards controller-aided multimedia dissemination in Named Data Networking," *2015 IEEE International Conference on Multimedia and Expo Workshops, ICMEW 2015*, 2015.

[87] W. Xiulei, C. Ming, H. Chao, W. Xi, and X. Changyou, "SDICN: A software defined deployable framework of information centric networking," *China Communications*, vol. 13, no. 3, pp. 53–65, 2016.

[88] J. Torres and O. Duarte, “An Autonomous and E ffi cient Controller-based Routing Scheme for Networking Named-Data Mobility,” *Tech. rep., Electrical Engineering Program, COPPE/UFRJ (April 2016)*, 2016.

[89] P.-C. Lin, Y.-D. Lin, T.-H. Lee, , and Y.-C. others Lai, “Using string matching for deep packet inspection,” *Computer*, vol. 41, pp. 23—-+, 2008.

[90] A. F. R. Trajano and M. P. Fernandez, “ContentSDN: A Content-Based Transparent Proxy Architecture in Software-Defined Networking,” *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, no. 1, pp. 532–539, 2016.

[91] A. Chanda and C. Westphal, “Content as a Network Primitive,” *arXiv preprint arXiv:1212.3341*, pp. 47–48, 2012.

[92] D. Chang, M. Kwak, N. Choi, T. Kwon, and Y. Choi, “C-flow: An efficient content delivery framework with OpenFlow,” *The International Conference on Information Networking 2014 (ICOIN2014)*, pp. 270–275, 2014.

[93] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Mazza, G. Morabito, A. Araldo, L. Linguaglossa, and L. Veltri, “Supporting COntent NETworking in Software Defined Networks,” *2012). http: finetgroup. uniroma2. it/twiki/bin/view. cgi/Netgroup*, no. July, pp. 1–28, 2012.

[94] W. P. Stefano and S. Salsano, “Deliverable 9.3 EXOTIC final evaluation and overall report,” no. 46, pp. 1–46, 2013.

[95] T. Ren and Y. Xu, “Analysis of the New Features of OpenFlow 1.4,” *Proceedings of the 2nd International Conference on Information, Electronics and Computer*, no. Icieac, pp. 73–77, 2014.

[96] I. Carvalho, F. Faria, E. Cerqueira, and A. Abelem, “ContentFlow: An Introductory

Routing Proposal for Content Centric Networks using Openflow," *API, 7th Think-Tank Meeting*, 2012.

[97] F. Schneider and D. Kutscher, "A method for operating an information-centric network and network," *Google Patents, US Patent App. 14/917,579*, 2013.

[98] P. Li, M. Wu, N. Wang, and H. Liu, "Supporting Information-Centric Networking in SDN," *International Journal of Future Computer and Communication*, vol. 4, no. 6, pp. 386–390, 2015.

[99] J. Wang, W. Gao, Y. Liang, R. Qin, J. Wang, and S. Liu, "SD-ICN: An interoperable deployment framework for software-defined information-centric networks," *Proceedings - IEEE INFOCOM*, pp. 149–150, 2014.

[100] A. Orebaugh, G. Ramirez, and J. Beale, "Wireshark & Ethereal network protocol analyzer toolkit," *Syngress*, 2006.

[101] J. W. Shipman, "Tkinter reference: a GUI for Python," *New Mexico Tech Computer Center, Socorro, New Mexico*, 2010.

[102] Y. Nakagawa, K. Hyoudou, and T. Shimizu, "A management method of IP multicast in overlay networks using openflow," *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, p. 91, 2012.

[103] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang, "A Concise Forwarding Information Base for Scalable and Fast Flat Name Switching," 2016.

[104] N. Handigol and S. Seetharaman, "Aster* x: Load-balancing as a network primitive," *9th GENI Engineering Conference*, pp. 1–2, 2010.

[105] M. Pa, "Installing and Using PyPy Standalone Compiler with Parlib Framework Technology Overview and Installation Manual," 2012.

[106] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a LLVM-based Python JIT compiler," *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, p. 7, 2015.