ALGORITHMS AND TECHNIQUES FOR TRANSITIONING TO SOFTWARE DEFINED NETWORKS

By

Prithviraj Patil

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August, 2016

Nashville, Tennessee

Approved:

Aniruddha S. Gokhale, Ph.D.

Akram Hakiri, Ph.D.

Parameshwaran Krishnan, Ph.D.

Gautam Biswas, Ph.D.

Shivakumar Sastry, Ph.D.

To my parents, Aai and Baba

ACKNOWLEDGMENTS

First and foremost, I would like to express my special gratitude to my advisor Dr. Aniruddha S. Gokhale, for providing me supervision, advice, guidance, and continuous support over the past six years. He made these six tough years of my Ph.D. studies and research fun with his friendship and humor. I am deeply grateful to him for always being there for research discussions and advice, and so thankful to him for his tireless efforts and time he spent on improving all my research works and research papers.

I would like to thank Dr. Akram Hakiri, Dr. P Krishnan, Dr. Gautam Biswas, Dr. Shiva Sastry for agreeing to serve on my dissertation committee and providing me with valuable feedback on my research work. I am especially grateful to Dr. Akram Hakiri for numerous discussions he had with me and also for valuable insights and feedback he has provided me about my work.

My research has been supported by various agencies and this work would not have been possible without the financial support from them. I would like to thank to the Defense Advanced Research Projects Agency (DARPA), Air Force Research Lab (AFRL), National Science Foundation (NSF), and the Air Force Office of Scientific Research (AFOSR).

I appreciate the feedback I received from past and present Distributed Object Computing group members: Akshay Dabholkar, James Edmondson, Kyoungho An, Subhav Pradhan, Faruk Calgar, Shashank Shekhar, Shweta Khare, Yogesh Barve, Shunxing Bao, and Anirban Bhattacharjee. I would especially like to thank my close friends, Nilesh Patil and Indranil Chatterjee, for providing moral support during these years.

Finally, I would like to acknowledge my parents (Aai and Baba), brother (Dada) and sister-in-law (Vahini) for being there for me in all the weathers. I could not have reached this stage without their support.

ABSTRACT

Software Defined Networking (SDN) has seen growing deployment in the large wired data center networks due to its advantages like better network manageability and higher-level abstractions. At the core of SDN is the separation and centralization of the control plane from the forwarding elements in the network as opposed to the distributed control plane of current networks. However various issues need to be addressed for an efficient transition to SDN from existing legacy networks. In this thesis, we address following three important challenges in this regards. (1) The task of deploying the distributed controllers continues to be performed in a manual and static way. To address this problem, we present a novel approach called InitSDN to bootstrapping the distributed software defined network architecture and deploying the distributed controllers. (2) Data center networks (DCNs) rely heavily on the use of group communications for various tasks such as management utilities, collaborative applications, distributed databases, etc. SDN provides new opportunities for re-engineering multicast protocols that can address current limitations with IP multicast. To that end we present a novel approach to using SDN-based multicast (SDMC) for flexible, network load-aware, and switch memory-efficient group communication in DCNs. (3) SDN has been slow to be used in the wireless scenario like wireless mesh networks (WSN) compared to wired data center networks. This is due to the fact that SDN (and its underlying OpenFlow protocol) was designed initially to run in the wired network where SDN controller has wired access to all the switches in the network. To address this challenge, we propose a pure opneflow based approach for adapting SDN in wireless mesh netowrks by extending current OpenFlow protocol for routing in the wireless network.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

ABSTRACT iv

LIST OF TABLES viii

LIST OF FIGURES ix

Chapter

I. Introduction 1

I.1. Software Defined Networking 1
I.2. Challenges and proposed Solutions for Software Defined Networks 3
I.2.1. Distributed Control Plane Management 3
I.2.2. Group Communication in Date Center Networks using SDN 4
I.2.3. Software defined Wireless mesh networks 5

II. Bootstrapping Software Defined Network for Flexible and Dynamic Control Plane Management using InitSDN 6

II.1. Motivation 6
II.2. Problem Description 8
II.2.1. Control Plane Message Types 8
II.2.2. Limitations of Existing Control Plane 10
II.2.3. Problem Statement 11
II.3. Solution Approach 11
II.3.1. Intuition Behind our Solution 12
II.3.2. InitSDN in Action 15
II.3.3. Implementation Details for Prototype 17
II.4. Experimental Evaluation 19
II.4.1. Evaluation Criteria: Building Network Applications for SDN Control Plane Management 19
II.4.2. Evaluation Criteria: Controller Scale-up/Scale-down . . 20
II.4.3. Evaluation Criteria: Controller/switch Migration 21
II.4.4. Evaluation Criteria: Managing Control-Plane Topology of Virtual SDNs 21
II.5. Related Works 22

II.6. Concluding Remarks 24

III. SDN-based Adaptive Multicast (SDMC) For Efficient Group Communication in Data Center Networks 26

III.1. Motivation 26
III.1.1. Importance of Group Communication in Large Data Centers 26
III.1.2. IP Multicast for group communication in DCN and its limitations 28
III.1.3. SDN based multicast 28
III.1.4. Our Contribution 29
III.2. Problem Description 29
III.3. Solution Approach 32
III.3.1. SDMC infrastructure 32
III.3.2. Lazy Initialization 35
III.3.3. Two-level SDMC-ID 36
III.3.4. Network link monitoring : 37
III.3.5. Network Switch-memory monitoring: 37
III.3.6. Workings of SDMC: 38
III.4. Experimental Evaluation 46
III.4.1. Average latency for receivers: 47
III.4.2. Network Load Adaptive-ness: 48
III.4.3. Switch-Memory Utilization Adaptive-ness: 48
III.4.4. Packet Loss 49
III.5. Related Works 49
III.6. Concluding Remarks 51

IV. Software defined Wireless mesh networks 52

IV.1. Introduction 52
IV.1.1. Wireless Mesh Networks (WMN) 52
IV.1.2. Software Defined Wireless Mesh Networks (SD-WMN) 53
IV.1.3. Contributions & Outline 54
IV.2. Desgin 55
IV.2.1. Motivation behind Three-Stage Routing 55
IV.2.2. Existing Solutions 57
IV.2.3. Design considerations 58
IV.3. Three-Stage Routing: Architecture 58
IV.3.1. OpenFlow Modifications for staged routing 61
IV.4. Experimental Evaluation 63
IV.5. Conclusions 66

V. Summary 69

V.1. Summary of Contributions 69
V.2. Summary of Publications 70

REFERENCES 72

LIST OF TABLES

Table Page

1. Network Link Monitoring 38

2. Network Switch Memory Monitoring 39

LIST OF FIGURES

Figure Page

1. Software defined networking layers 2

2. Three Types of Messages Flowing in the Distributed Controller Architecture 10

3. InitSDN modular architecture 14

4. Legacy Network During the Bootstrapping Phase (1) network slicing step has been executed (2) Brown colored hosts are chosen to be in the control plane as per topology and configuration 16

5. Legacy Network turned into SDN Network After Bootstrapping is Completed (1) InitSDN has taken a back seat (2) SDN controllers are placed in control plane, configured and have been activated 18

6. Three Types of Messages Flowing in the bootstrapped SDN 19

7. SDMC infrastructure 33

8. Initial SDMC Sender Setup 36

9. Initial SDMC Receiver Setup 37

10. Adapting to Network Load 42

11. Adapting to Switch Memory Utilization 44

12. Receiver Latency against group size 47

13. Receiver Latency for different network topology and size 48

14. SDMC adaptive-ness to network load and to switch memory utilization . 49

15. SDMC Packet loss 50

16. SDN Based Wireless Mesh Network 55

17. Hybrid Architecture for SDN Wireless Mesh Network (1) 56

18. Hybrid Architecture for SDN Wireless Mesh Network (2) 57

19. Example SDN Mesh Scenario 59
20. Stage I Interactions 59
21. Stage II Interactions 60
22. Stage III Interactions 60
23. Implementation components 63
24. Controller Switch Connection Latency 65
25. Controller Switch Re-Connection Latency 67
26. Switch-Switch Connection Latency 67

CHAPTER I

INTRODUCTION

I.1 Software Defined Networking

The control plane in the network devices has historically been tightly coupled with the data plane. Although this approach has the benefit of promoting an inherently distributed architecture, it makes it difficult to manage, program, update and upgrade the control plane without impacting the data plane. Overcoming these difficulties has led to the vision of programmable networks with Software Defined Networking (SDN) [4, 17] being a front-runner among the emerging solutions. The primary idea behind SDN is to move the control plane outside the switches and enable external control of data plane through a logical software entity called controller. The controller offers northbound interfaces to network applications and southbound interfaces to communicate with data plane. OpenFlow is one of the possible southbound protocols.

At the core of SDN is the separation and centralization of the control plane from the forwarding elements in the network as opposed to the distributed control plane of current networks. This decoupling allows deployment of standards-based software abstraction between the network control plane – the so called SDN controller – and the underlying data plane, including both physical and virtual devices. This standards-based data plane abstraction, called OpenFlow, provides a novel approach to dynamically provision the network fabric from a centralized software-based controller. SDN architecture envisions a centralized control plane, which may result in adverse consequences to the reliability and performance [9]. First, grouping all the functionality into a single node requires more computation power, data storage and throughput to deliver the traffic. Second, a centralized software controller will incur higher packet processing latency due to increased traffic and can become a single point of failure. For example, as the size of data centers and cloud

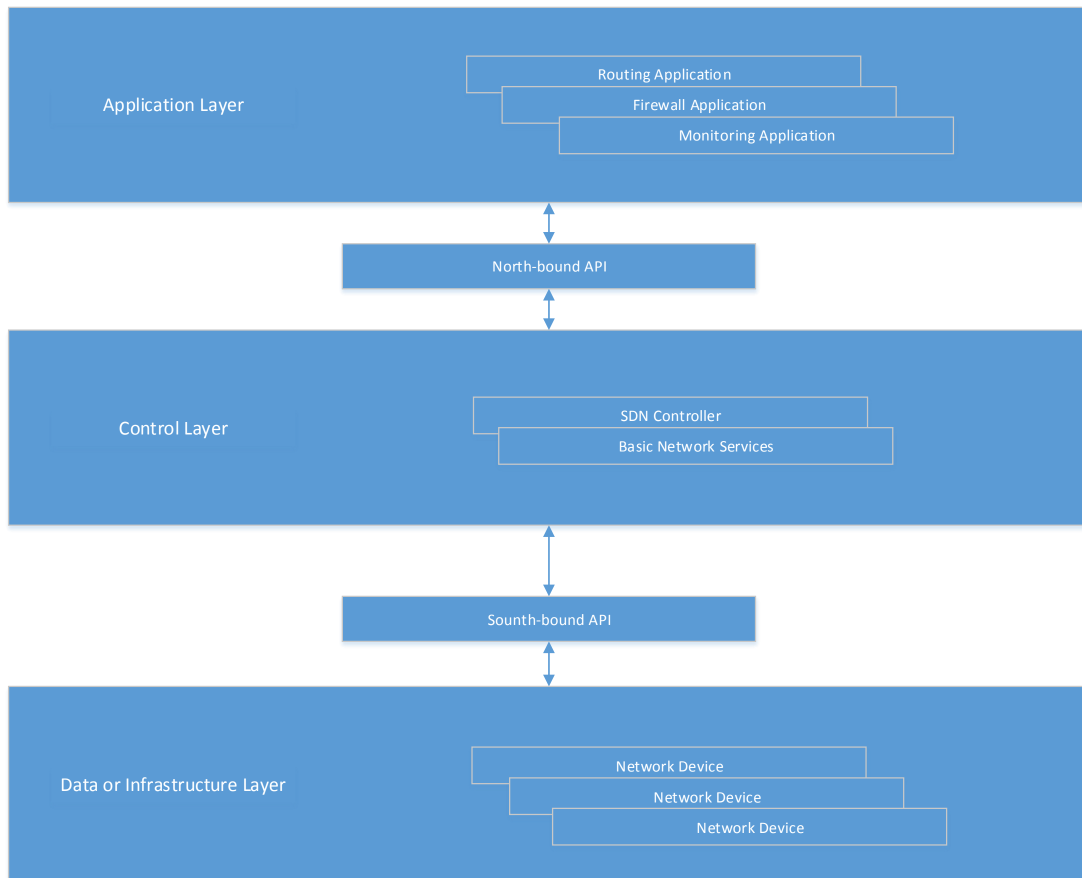

Figure 1: Software defined networking layers

computing networks increase to thousands or even millions of nodes in the near future, the speed up of the controller is a critical issue where neither the over-provisioning mechanisms nor load-balancing solutions can solve the scalability problems.

In this way, Software-Defined Networking (SDN) has emerged as a new intelligent architecture for network programmability. It moves the control plane outside the switches to enable external centralized control of data through a logical software entity called controller. The controller offers northbound interfaces to network applications that provide higher level abstractions to program various network-level services and applications. It also uses southbound interfaces to communicate with network devices. OpenFlow is an example of southbound protocols. OpenFlow behavior is simple but it can allow complex

configurations: the hardware processing pipeline from legacy switches is replaced by a software pipeline based on flow tables. These flow tables are composed of simple rules to process packets, forward them to another table and finally send them to an output queue or port. One complementary technology to SDN called Network Function Virtualization (NFV) has the potential to dramatically impact future networking by providing techniques to re-factor the architecture of legacy networks by virtualizing as many network functions as possible. NFV advocates the virtualization of network functions as software modules running on standardized IT infrastructure (like commercial off-the-shelf servers), which can be assembled and/or chained to create services

Software Defined Networking (SDN) has seen growing deployment in the large wired data center networks due to its advantages like better network manageability and higher-level abstractions. However various issues need to be addressed for an efficient transition to SDN from existing legacy networks. In this thesis, we address following three important challenges in this regards.

I.2 Challenges and proposed Solutions for Software Defined Networks

I.2.1 Distributed Control Plane Management

To improve reliability and performance of Software Defined Networking (SDN) architectures, a number of recent efforts have proposed a logically centralized but physically distributed controller design that overcomes the bottleneck introduced by a single physical controller. Despite these advances, two key problems still persist. First, the task of controlling the host network and the task of controlling the control-plane network remain tightly intertwined, which incurs unwanted complexity in the controller design. Second, the task of deploying the distributed controllers continues to be performed in a manual and static way.

Solution: Bootstrapping Software Defined Network for Flexible and Dynamic Control Plane Management using InitSDN: To address these two problems, this work presents a

novel approach called InitSDN to bootstrapping the distributed software defined network architecture and deploying the distributed controllers. InitSDN makes the SDN control plane design less complex, makes coordination among controllers flexible, provides additional reliability to the distributed control plane.

I.2.2 Group Communication in Date Center Networks using SDN

Data center networks heavily rely on the use group communication for various tasks. Data center management utilities (e.g. software update/upgrade, log management, resource monitoring, scaling various resources up or down, access control etc.), collaborative applications like social media, project management tools, version control systems etc.), multimedia applications, multi-player games are few examples of tasks that require efficient group communication. However though multicast is useful for efficient u group communication, IP multicast has seen very low deployment in the data center networks due to its deficiencies like inefficient scaling, inefficient switch-memory utilization, initial receiver latency. With the advent of SDN (Software defined Network), though, multicast has become easy to implement in the SDN controller, it too faces challenges faced by IP multicast.

Solution: SDN-based Adaptive Multicast(SDMC) For Efficient Group Communication in Data Center Networks: To address these problems, in this work, we propose an new way of implementing multicast protocol for group communication protocol in SDN enabled networks specifically in large data centers networks. This multicast protocol will be more lightweight, dynamic, adaptive to networking resources like link utilization and switch memory when compared to the traditional multicast solutions. We list our contributions in this work as below,

- Design a network-load-adaptive and switch-memory-adaptive multicast for data center networks .
- Implement it as a SDN network application running on the top of SDN controller.

- Evaluate it for different data-center network load variations, switch-memory utilization.

I.2.3 Software defined Wireless mesh networks

Software Defined Networking (SDN) has seen growing deployment in the large wired data center networks due to its advantages like better network manageability and higher-level abstractions. SDN however has been slow to be used in the wireless scenario like wireless mesh networks (WSN). This is due to the fact that SDN (and its underlying OpenFlow protocol) was designed initially to run in the wired network where SDN controller has wired access to all the switches in the network. Various workarounds have been proposed for adapting SDN and Openflow to the wireless setting. However all these approaches require some kind of hybrid switching hardware and software (especially for routing) which goes against the fundamental SDN architecture and also causes unnecessary increase in hardware and software complexity of the switch.

Solution: Three Stage Routing Protocol for SDN based Wireless Mesh Networks: To address this challenge, we propose a pure opneflow based approach for adapting SDN in wireless mesh netowrks by extending current OpenFlow protocol for routing in the wireless network. We describe the extension to OpenFlow protocol and also its use in a novel three stage routing strategy which allows us to adapt a centralized routing of SDN in an inherently distributed wireless mesh network without requiring additional support from switch hardware. We evaluate our approach with the existing hybrid approach using latency metric for controller-switch and switch-switch connections.

CHAPTER II

BOOTSTRAPPING SOFTWARE DEFINED NETWORK FOR FLEXIBLE AND DYNAMIC CONTROL PLANE MANAGEMENT USING INITSDN

To improve reliability and performance of Software Defined Networking (SDN) architectures, a number of recent efforts have proposed a logically centralized but physically distributed controller design that overcomes the bottleneck introduced by a single physical controller. Despite these advances, two key problems still persist. First, the task of controlling the host network and the task of controlling the control-plane network remain tightly intertwined, which incurs unwanted complexity in the controller design. Second, the task of deploying the distributed controllers continues to be performed in a manual and static way. To address these two problems, this work presents a novel approach called InitSDN to bootstrapping the distributed software defined network architecture and deploying the distributed controllers. InitSDN makes the SDN control plane design less complex, makes coordination among controllers flexible, provides additional reliability to the distributed control plane.

II.1 Motivation

Recent efforts have proposed a logically centralized but physically distributed control plane [9]. The distributed control plane is more responsive to handle network events because the controllers tend to be closer to the events than the centralized architecture. However, these solutions incur a different set of complexities for developing and managing the controllers. One key limitation of these approaches is that they club task of controlling the host network and task of managing the distributed control plane together. There is no global optimal view of the network to keep a consistent network state among multiple controllers. Additionally, increasing the number of controllers does not necessarily guarantee a linear

scale up of the architecture nor does it improve the flexibility or enhance the performance. Hence, the developer of a distributed controller now has to take care of all the concerns that arise out of distributed nature of the system including controller synchronization, controller replication, controller logic partitioning and controller placement [12, 36]. All the above issues are orthogonal to the fundamental controller functionality. However current distributed control plane architecture forces controller developer to invest energy into addressing these issues which complicates the controller design and management and makes control-plane inflexible.

To address these problems, we propose a solution called *InitSDN,* which is based on a bootstrapping mechanism that helps to decouple the orthogonal distributed systems concerns from the primary issues related to the controller. InitSDN is designed to make SDN more flexible, reliable, fault-tolerant without adding complexity to the controllers.

InitSDN divides a single physical network substrate into two slices: a *dataslice* for controlling the hosts that run user applications and a *controlslice* for controlling the controllers. Based on the configuration or strategy defined by a network operator, InitSDN allocates the right number of hosts between these two slices,[1] selects an initial topology for the *controlslice*, deploys required controllers in the *controlslice*, sets the coordination mechanism among the controllers, maps the switches in the *dataslice* to distributed controllers, and kick-starts the operation of the real/actual SDN. Over the course of the SDN operation, InitSDN can increase or decrease the size of slices dynamically, change the topology of the *controlslice*, change the coordination mechanism among the controllers (e.g. use Zookeeper or Chubby, etc) to adapt to network topology changes or to dynamic network loads or simply as part of an upgrade.

[1] In a shared or in-band control network, which is our focus, the controller logic must reside on some host of that network and hence some hosts will be used for hosting the controller logic while others will be used for application logic.

II.2 Problem Description

In this section, we describe problem desciption of distributed controller placement problem.

II.2.1 Control Plane Message Types

We categorize messages that are being exchanged in the SDN in three different categories as described below:

1. *Control messages:* These are the messages that are used to control the communication between the hosts. It includes various OpenFlow messages like OFPT_FLOW-MOD, OFPT_FLOW_REMOVED, OFPT_PACEKT_IN etc. These messages flow between controller-switch pairs.

2. *Data messsages:* These are normal data packets sent/received by hosts. These messages normally flow between switch-host or switch-switch pairs.

3. *Meta-control messages:* We define meta-control messages as those messages that are used to control the communication between SDN infrastructure entities, i.e. controllers, switches. It includes all the messages that are required for controller-switch connection setup, connection tear-down, controller-migration, switch-migration, host-migration, network discovery and topology services, controller logic synchronization or backup, etc. These messages flow between controller-switch, controller-controller and switch-switch pairs. It can includes OpenFlow messages like OFPT_FLOW-MOD, OFPT_FLOW_REMOVED, OFPT_PACEKT_IN, OFPT_PACKET_OUT etc. Also in addition to above OpenFlow messages, it may include other non-OpenFlow non-standardized messages and different solutions may implement them in their own proprietary manner.

Figure 2, shows three kinds of messages flowing in the SDN network: (1) control messages (shown as red pipes), (2) data messages (shown as yellow pipes) and (3) meta-control messages (shown as blue pipes). We illustrate distinction between these three messages using couple of examples below. Suppose, initially there are no flow rules installed in any of the switches of Figure 2. We use notations (P1,P2,P3,...) for numbering packets; (H1,H2,H3,...) for numbering hosts; (S1,S2,S3,...) for numbering switches and (C1,C2,C3,...) for numbering controllers.

1. *Example 1:* H1 sends a P1 (e.g. TCP) to H2. P1 reaches S1. S1 does not have flow rule to handle it. Hence it sends P1 enclosed in the OpenFlow PACKET_IN message P2 to the C1. C1 then sends either OpenFlow FLOW_MOD message P3 or PACKET_OUT message P4 to S1. In this example, P2, P3, P4 are control messages since they deal with the control of host network i.e. (network between H1 and H2). And P1 is a data message since it is the part of host network.

2. *Example 2:* C1 wants to find the network topology. To do that, it sends OpenFlow PACKET_OUT message P1 encapsulating LLDP broadcast message P2 in it to all its connected switches. All connected switches receive this message. Lets focus on S1 only. On receiving P1, S1 modifies it and adds its Link Layer address as Source L2 address in P2. Lets call this new packet P3. S1 then broadcasts P3. All switches reachable from S1 will receive P3 while unreachable switches will not. These reachable switches do not have flow rule to handle P3. Hence they will send P3 enclosed in OpenFlow PACKET_IN message P4 to C1. In this way, C1 can now build a topology of all the switches reachable from S1. In this example, P1, P2, P3, P4 are all meta-control messages since they deal with the control/management of the control network. Note that some of these messages are OpenFlow while some are non-OpenFlow.

As illustrated by above examples and by Figure 2, different types of messages deal with

different concerns in the SDN. But existing distributed control-plane does not make distinction among them, which becomes problematic as explained next.

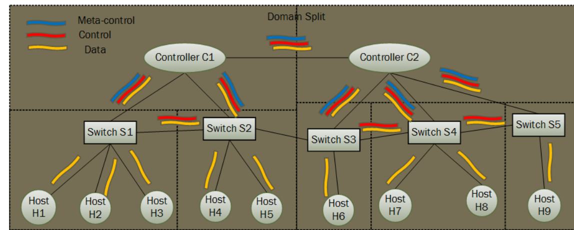

Figure 2: Three Types of Messages Flowing in the Distributed Controller Architecture

II.2.2 Limitations of Existing Control Plane

A number of prior studies have proposed designs for a distributed, scalable, and fault tolerant controller architecture in the SDN [5, 9, 12]. A key commonality across these approaches is to add a connection management module in the controller alongside the Openflow module. This module is responsible for tasks like leader election, synchronization, participation in switch migration, managing backups, state consistency, etc.

There are two basic problems with such distributed control plane design. First, in such architectures, the data messages flow in the SDN network but control messages flow in the non-SDN legacy network. This occurs because currently, control messages need to be exchanged to set up the SDN first. Then only after SDN is setup (i.e. switches are configured with correct controller references and flow-rules), data messages can be exchanged. Hence control messages are thought to be flowing in the pre-SDN (or non-SDN or legacy network).

This design is sufficient for supporting a subset of network applications like routing or network monitoring where applications need to instrument only data messages and not control messages. However, many other network applications exist that perform load-balancing, leasing virtual network switches using network hypervisors (e.g. flowvisor [30],

OpenVirteX [3]), mobile networking and others which require instrumentation of both the data and control messages. Some example use cases that demonstrate these needs are: (1) in mobile networking, switches and controllers migration requires instrumentation of control messages (2) for applications like leasing virtual network switches or load balancing where dynamic controller placement is needed that is based on control message instrumentation. Using existing control plane architecture, the above applications need to operate in both the SDN network (for data messages) and non-SDN legacy network (for control messages) simultaneously.

Secondly, in these architectures, the control and meta-control messages are clubbed together, i.e. they originate from the same controller. This forces the controllers to handle many of the distributed system complexities, such as handling partitioning, placement, consensus, synchronization, coordination, which complicates the design of the controller and violates many of the software engineering principles resulting in code that is hard to maintain and evolve.

II.2.3 Problem Statement

Controller should perform only core controller functionality (i.e. to control communication between the host network). Any other complexity that arises due to the distributed nature of the controller design should be stripped away from controller to make controller design simpler and modular. Also network application should be able to instrument both the data messages and control messages exclusively inside the SDN , without any help from legacy network.

II.3 Solution Approach

We now discuss the design and implementation details of InitSDN. We start with the key idea behind InitSDN, its architecture, message flow and a use case illustrating how InitSDN can be used.

II.3.1 Intuition Behind our Solution

As discussed in Section II.2, the SDN control plane faces two key problems:

1. *A chicken-and-egg situation:* In the current control plane design, data messages flow in the SDN network but control messages flow in the non-SDN legacy network. This is because of the classic chicken-and-egg situation where we need a way to exchange control messages before starting to exchange any data message in SDN. Hence, control messages tend to operate in the legacy network as opposed to the SDN network.

2. *Separation of concerns in the SDN:* In the current control plane, control and meta-control messages originate from the same controller. From the definition of the message types, we can articulate that the data and control messages form the operational concern of SDN while the meta-control messages form the initialization concern of SDN. Hence, clubbing these together renders the control plane inflexible.

In searching for a solution to these problems, we realized that *bootstrapping* is a very basic and fundamental concept used in computational systems that addresses the chicken-egg scenario and as a way to achieving separation of initialization and operational concerns. A classic example of bootstrapping is found in operating systems boot loading where the loading of the OS is done by the boot-loaders like grub or lilo. It helps to relieve the operating system from the burden of booting and dealing with BIOS related issues which are accidental/orthogonal to the objectives of the OS.

In the context of OS, the mounting of the root file system is another scenario where bootstrapping is a necessity. The Initramfs (earlier called as initrd) is used to search for the root file system which may reside on the hard disk, removable disk or on the network. It then mounts the root file system and hands over the control to it. The root file system is responsible for all the further disk I/Os. Without initramfs, it becomes a chicken-and-egg problem where an OS is supposed to search and mount a root file system which in turn requires the root file systems to be mounted first.

The reason bootstrapping is able to solve these problems is because it essentially confines/restricts the scope of staticness, inflexibility and tight-coupledness that is present in the system to a very small portion of system instead of permeating it into the entire system. Since SDN faces the same two problems that are faced by operating systems and root file systems, we can exploit the bootstrapping pattern in the SDN during its initialization so that the static-ness and inflexibility is confined to a small part of the network allowing SDN to become more flexible, dynamic, scalable and hence more reliable. Therefore we have designed a solution by adding an initialization phase in the SDN setup for bootstrapping called as InitSDN (for initialization of SDN).

We now present the architecture and implementation details of the InitSDN approach. InitSDN works in two phases. The first phase is the initialization (or bootstrapping) phase and second phase is starting the real (or root) SDN network. In the first phase, a statically configured SDN is started and is responsible for loading (or booting) the real (or root) SDN with all the required controllers, switches and applications with the appropriate number of controllers, topology of controllers, type of controllers (standalone, distributed, hierarchical etc.), and communication protocols for meta-control messages on the physical network substrate, which are all determined and set by the network operator.

Figure 3 shows the architecture of InitSDN. It works in the legacy network (i.e., non SDN) that uses the TCP/IP protocol. InitSDN has a modular structure with various modules as follows:

1. *Network discovery & topology service:* This is the basic module of the InitSDN. It discovers the switches and hosts in the network. It then creates the model of the network topology using specialized packets. It sends LLDP (Link Layer Discovery Protocol) packets to switches, parses the reply messages and builds the topology model.

2. *Network Hypervisor:* This module provides access to the existing network hypervisors. A network hypervisor is used to slice the network into control and data slice.

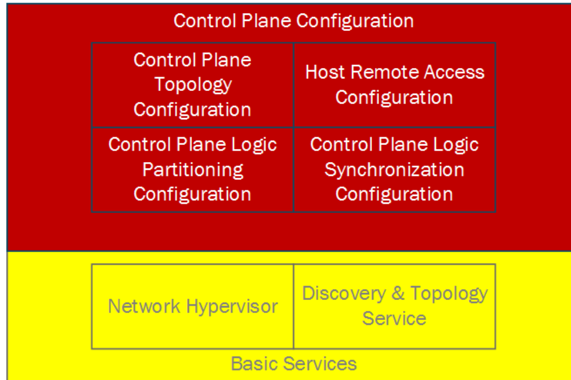

Figure 3: InitSDN modular architecture

Currently we have used Flowvisor [30]. This module is also used to create virtual switches for multi-tenant network applications. For this, currently we use OpenVirtex [3]. However, our design can accommodate other network hypervisors.

3. *Control-plane topology:* This module allows the network operator to specify the initial control plane topology. By default, InitSDN uses the basic topology with one centralized controller and one backup controller. Network operators however, can provide their own control plane topology as described below. This module then slices the network into two slices using information from the previous two modules (i.e., network hypervisor configuration and discovery & topology service).

4. *Control-plane partitioning:* This module is used to slice the control plane logic. This requires the controller to expose an API to perform this action. These APIs currently are controller-specific. In our present implementation, we have used a modified POX controller. For example, Pyretic [28] has a modified POX client, which allows us to specify the flows to be controlled by the POX controller using a command line argument when starting the controller.

5. *Control-plane synchronization:* This module is used to specify the synchronization mechanism to be used in the control plane, e.g., how to synchronize the backup controller. Currently with the modified POX controller, we use Apache Zookeeper

for synchronization. The modified POX controller writes its state (e.g. topology, counter etc) to a file. This file then gets synchronized across the control plane. This module allows an operator to use any other synchronization mechanism, e.g., Vagrant Serf, Google Chubby, etc.

6. *Host Remote Access:* Since InitSDN installs controllers on the hosts, it needs access to do so on those hosts. This module provides a way to configure such access. At present this module uses a combination of SSH and SCP through the Python command line tool Fabric [26]. However, based on the host access policy, the network operator can use any other tool.

II.3.2 InitSDN in Action

Now we describe the steps involved in the booting of a legacy network into a flexible, dynamic and fault tolerant SDN network using InitSDN.

1. *Initial Setup:* We assume a network substrate which uses a legacy network with Openflow-enabled switches. InitSDN has remote access to all the hosts that are supposed to host the control plane. The chosen SDN controller exposes the API to configure the partitioning and synchronization strategy.

2. InitSDN is started on one of the hosts in this network substrate and is connected to all (top-level) main switches statically.

3. An InitSDN network application will then configure the InitSDN controller. This InitSDN application contains configuration information of all the InitSDN control-plane modules shown in Figure 3 and also described in the previous Section II.2.

4. InitSDN will build a model of the topology of the network using the discovery and topology module. The topology contains all the hosts, switches and links present in

the network. It will also contain link properties and switch configurations like the ones supported in the OpenFlow version.

5. InitSDN then builds the control-plane topology based on the configuration provided by the network operator and network topology model from the previous step.

6. Using the network hypervisor (e.g. flowvisor), InitSDN will slice the network into two slices namely data-slice and control-slice. The number of hosts in both the slices and their topology is determined by the control-plane topology from the previous step.

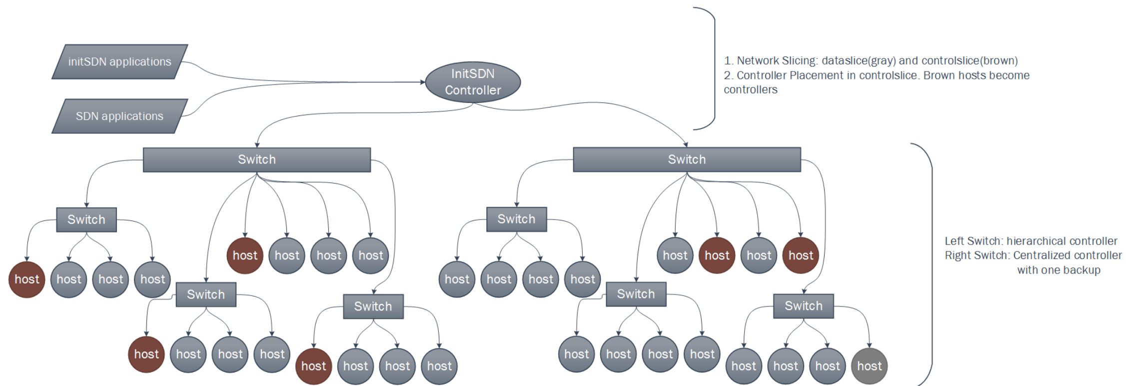

Figure 4: Legacy Network During the Bootstrapping Phase (1) network slicing step has been executed (2) Brown colored hosts are chosen to be in the control plane as per topology and configuration

7. InitSDN then remotely installs the controller in all the hosts in the control plane.

8. InitSDN configures the controllers in the control plane as per the control plane partitioning strategy provided by the network operator, e.g., controller C1 handles only secure flows while controller C2 handles only non-secure flows, etc.

9. InitSDN configures the synchronization strategy in the control plane as the controllers need to share the local topology changes with the other (non-local or remote)

controllers, e.g., backup controllers need to be synchronized with the respective primary controller, etc.

10. InitSDN then installs the default flow-rules in the switches so that in case of control plane failure, switch will notify InitSDN. This adds an additional level of reliability to the SDN control plane.

11. InitSDN then configures all the switches with one or more controllers from the control-slice.

12. At this point, SDN is considered to be booted as per the configuration provided by the network operator and InitSDN is out of the picture.

II.3.3 Implementation Details for Prototype

The following tools and technologies were used to realize InitSDN and evaluate its properties.

- *Network Emulation:* Mininet [19].
- *Switch:* OpenVswitch and Openflow's Reference Switch (ofdatapath) [24].
- *Controller:* Openflow's Reference Controller [24], Apache Floodlight, Stanford University's Pox and Ryu.
- *Host:* Docker Containers and VirtualBox VMs.
- *Network Virtualization:* Flowvisor [30], OpenVirtex [3].
- *Network Topologies:* Real network topologies (built using traceroute) obtained from Stanford University [32, 33].
- *Distributed Consensus and Synchronization:* Hashicorp Serf, Apache ZooKeeper, Google Chubby, Doozerd, etc.

- *Host Remote Access:* Fabric SSH

We used Mininet to simulate the real world communication network with hosts, switches and controllers. We have configured the Mininet to use OVS user space switch and reference switch (ofdatapath) provided by the Openflow. We have implemented InitSDN on top of the network virtualizer Flowvisor and OpenVirtex. Flowvisor is used to slice the network into control-slice and data-slice. OpenVirtex is used to create multiple virtual SDNs(vSDNs) with full address space and own topology. POX is used for building logically centralized but physically distributed control-plane. Since the default hosts provided by the Mininet do not have full isolation, we use docker containers and/or VirtualBox virtual machines to serve as hosts. For distributed consensus and synchronization among the controllers we use Apache Zookeeper. We also use few other tools like Hashicorp Serf, Google Chubby just to show the flexibility of InitSDN. For evaluation purpose we used various real world internet scale network topologies from the Stanford University.

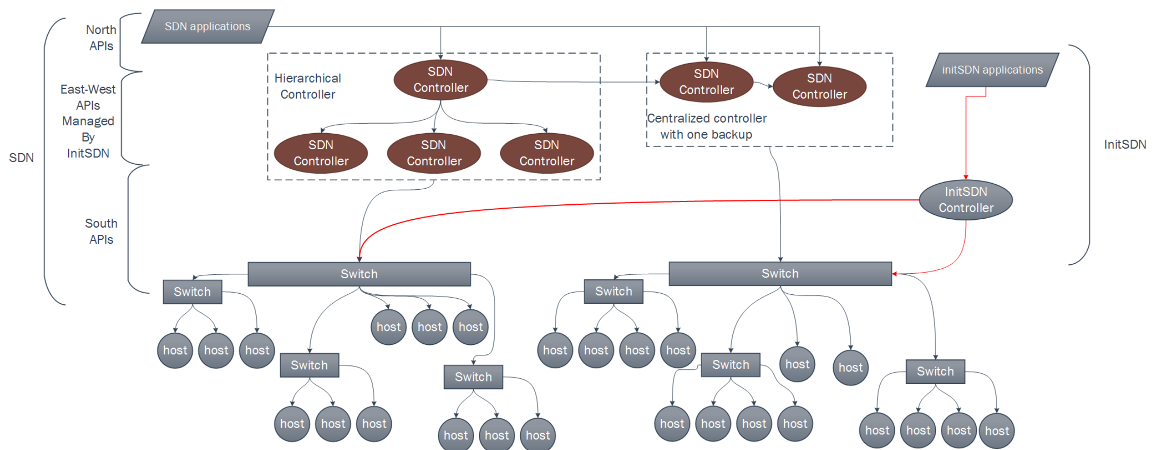

Figure 5: Legacy Network turned into SDN Network After Bootstrapping is Completed (1) InitSDN has taken a back seat (2) SDN controllers are placed in control plane, configured and have been activated

II.4 Experimental Evaluation

In this section we provide a qualitative evaluation of InitSDN's capabilities. In evaluating InitSDN qualitatively we focus on properties such as the ease of performing some of general use cases for the management of SDN control plane with and without InitSDN.

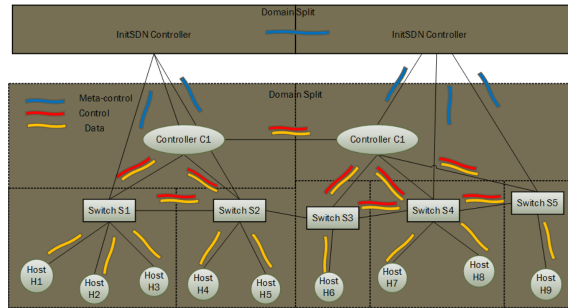

Figure 6: Three Types of Messages Flowing in the bootstrapped SDN

II.4.1 Evaluation Criteria: Building Network Applications for SDN Control Plane Management

This criteria is relevant to the SDN service providers. As we discussed in the previous section, InitSDN separates the control and meta-control messages as shown in Figure 6. This helps to modularize the network applications by providing separation of concerns between two different types of applications as follows:

1. *SDN network application:* These are the network applications that instrument the network among the hosts. These are developed by the SDN user or vSDN(virtual SDN) tenant. Examples of such applications are routing (OSPF, IS-IS, BGP etc), security, access control, application-based forwarding, etc. These applications are written against the controller that client is using in its SDN (or vSDN).

2. *InitSDN network application:* There is another type of application that instruments the network along the control-plane. These are developed by the SDN service providers.

Examples of such applications are switch migration, controller migration, VM network state migration, control-plane scale up/down, controller updates, control-plane topology management, vSDN control-plane management, etc. Without InitSDN, these applications have to be written for individual controllers. For example, if SDN hosts three types of controllers, then the controller migration application has to be written for each of these controllers. However, these applications become easier to develop with InitSDN since such applications now need to be written against only InitSDN irrespective of the number of controllers, number of vSDNs, or types of controllers present in the system.

In this way, InitSDN brings the separation of concerns in the SDN control plane management.

II.4.2 Evaluation Criteria: Controller Scale-up/Scale-down

Controller scale-up or scale-down can be achieved easily using InitSDN.

1. *scale-up:* InitSDN needs to find out idle hosts (or VMs) for adding them to the control-plane. This has to be programmed by a network operator through the InitSDN application. InitSDN then adds such new hosts to the control plane. InitSDN installs controllers on these new hosts. It also modifies flow-rules on new switches, so that they start to redirect their traffic to new controllers.

2. *scale-down:* InitSDN simply modifies the flow rules in the switches to point them to controllers from to be scaled-down control-plane only. After that InitSDN can either shutdown hosts containing extra controllers (i.e. those controllers which are now not connected to any switches) or use them for other controllers (e.g. different vSDN).

This way InitSDN provides scalability to the SDN control plane. This also increases reliability of SDN control plane against network load changes.

1. Make InitSDN build a new topology.

2. Compare old and new topology and find out the scale up/down steps required.

3. Ask InitSDN to scale up/down accordingly.

II.4.3 Evaluation Criteria: Controller/switch Migration

In InitSDN, the controller or switch migration is reduced simply to the task of updating the control-plane topology. InitSDN builds new control-plane topology after notified by its discovery module about the change in the network topology. This new topology is then enforced on the control plane as described in the previous subsection.

II.4.4 Evaluation Criteria: Managing Control-Plane Topology of Virtual SDNs

In the multi-tenant network infrastructure (e.g. large data centers), SDN will need to provide services for creating virtual SDNs(vSDN). In such environments, individual vSDN will be managed by the different independent controllers and may require different control plane topology. For example, Client A and Client B lease the one vSDN each from the SDN service provider C. C hosts both the vSDN on the same physical network hardware. A wants hierarchical control-plane with POX controllers, while B wants centralized control-plane with RYU controller. We can accomplish such requirements easily using the InitSDN as follows.

1. *Case when vSDN is created before SDN bootstraps i.e. statically:* Say, we need to create two vSDNs with different control-plane topology. Network hypervisor first creates two vSDNs on the data slice. It will then inform InitSDN about switches and hosts used to create these two vSDNs. It will also provide control-plane topology requirements of these two vSDNs. InitSDN then can calculate the total number of controllers needed to satisfy requirements of both vSDNs. This calculation depends upon the switches used by the both vSDNs. InitSDN can optimize the number of

hosts required for control-plane of these two vSDNs based on switch sharing between them.

2. *Case when vSDN is created dynamically i.e. after the SDN is booted:* In this case, similar approach is used, however InitSDN need to use the existing control plane or scale up if required. This decision is based on whether control-plane requirement of new vSDN is met by the existing control-plane.

This way InitSDN provides flexibility to the SDN control plane management.

II.5 Related Works

This section compares related efforts and contrasts them with InitSDN. In [9], the authors present a solution for an elastic distributed controller for the SDN called ElastiCon. It is capable of providing logically centralized but physically distributed control mechanisms with reliability and scalability services like switch migration, load balancing, fault tolerance among other properties in the southbound API. It consists of autonomous controller nodes that coordinate among themselves to provide a consistent control logic for the entire network. Every switch connects to multiple controllers, one of which is the master while others are slaves. Each controller contains a core module similar to the basic centralized controller which is responsible for the control plane management. Every controller also contains another module which is responsible for distributed controller services like leader election, state synchronization, switch migration from one controller to another, etc. While these properties are much desired, as discussed in Section II.2.2, such an approach adds to the complexity of the controller by not separating control and meta-control messages. Moreover, this approach does not provide a configurable control plane topology, e.g. a case where the network operator may want only a centralized controller.

The authors in [18] propose a solution called the Pratyaastha control plane to address a related but different controller placement problem. Pratyaastha first partitions the SDN

application state into the lowest granularity possible so that it can be distributed across the controllers. Subsequently, based on the controller load, it decides the placement (in this case, reassignment) policy that maps the switches and application state to the out-of-band controller instances. This placement problem is different than ours where the controllers are mapped to the physical hosts. Hence, Pratyaastha does not require the network hypervisors since the control plane still resides on the dedicated network. Though, Prayaastha provides elasticity to the control plane in the case of changing controller load, it does not provide elasticity in the case of major network topology changes or large-scale failures in the initially assigned control plane physical hosts since the control plane physical nodes are still statically assigned.

The authors in [11] describe a two-level controller hierarchy called "Kandoo" with the lower-level controllers being responsible for handling the frequent events and short-lived flows, while top-level controllers handle the other flows. However, it is not flexible enough to adapt to the network topology and load, e.g. in the case where most (or all) of the network flows are long-lived. The authors in [12] discuss the placement problem in the control plane and observe that a single controller is sufficient for most of the use cases. However, it does not consider the use case that requires robust fault tolerance, virtual SDNs, multi-level controller hierarchy, etc where multiple controllers are needed and hence placement becomes more complex. Another effort [36] discusses the controller placement problem but in the context of the network load alone. It does not provide configurable control-plane topology. Difane [37] and DevoFlow [7] extend the switches data-path mechanisms to offload the central controller. To scale the topology with multiple rules, lower delay, higher throughput, and reduce the time required for installing new rules, Difane tries to partly flood forwarding decisions from the controller to an authority switch. DevoFlow introduced new mechanisms to dispatch important events to the control plane. Kandoo [11] addressed the same issues, but instead of extending the switch's data path, it removes control data functions close to the switch. Kandoo distinguishes two-level hierarchies for controllers:

local distributed controllers and a logically centralized root controller. The local controllers are deployed throughout the network to process events locally, each controls one or more switches in the network and they do not need the network-wide state. The root controller controls all local controllers to access to non-local network-wide state. If the root controller wants to install flow entries on a switch, it delegates the requests to its local controller. As a result, Kandoo offloads control applications over available resources without violating any requirements of control applications.

HyperFlow [31] provides the tradeoff between the centralized control while keeping scalability, by passively synchronizing the entire network views and delegates the decision making to individual controllers. Hyperflow uses Flowvisor to slice the network to several partitions and enable multiple controllers in the network, each manage a single slice. The global synchronization is done by creating a rendezvous point between locally selected events, so state changes are propagated using publish/subscribe messaging bus between the distributed controllers and the switches in different slices.

II.6 Concluding Remarks

In this work, we highlighted the limitations of the current SDN distributed control plane in terms of controller complexity, reduced flexibility, scalability and reliability. To address these concerns, we described a solution approach that involves a separate bootstrapping or initialization phase for the SDN network. Our solution is called InitSDN and its architecture involves a number of functionalities that relate to topology, discovery, synchronization, and placement. Our current work has qualitatively evaluated the benefits stemming from the work in terms of ease of developing the controller logic and operationalizing the SDN network for network operators using real world network toplogies. To use our approach at present, we need to add a few APIs, e.g., for control plane logic partition and synchronization, to the existing controllers individually. We are working on creating a generic API for these tasks so that any future controller can be used in our solution seamlessly. Also

as a future work, we plan to obtain comparative performance results of the InitSDN such as controller-switch latency, recovery time of controller for various network topology and configurations.

In the context of our InitSDN, we make the following three contributions in this work:

- We propose and describe the architecture of the InitSDN controller used for bootstrapping a real SDN network,
- We describe the implementation details of the InitSDN controller.
- We qualitatively evaluate the benefits of our approach in terms of separation of concerns, reduced complexity of the SDN controller, increased reliability and better management of control-plane using various motivating use cases.

CHAPTER III

SDN-BASED ADAPTIVE MULTICAST (SDMC) FOR EFFICIENT GROUP COMMUNICATION IN DATA CENTER NETWORKS

In this chapter, we describe our work related to the SDN based multicast solution for efficient group communication in data center networks. First we describe the organization of this chapter. Next, in the section III.1, we provide motivation and background information for IP based multicast and SDN based multicast. In the section III.2, we discuss the considerations that drive the SDMC design and its architecture. We also describe the problem statement in this section. In the section III.3, we describe the architecture and implementation of the SDMC. This section also provides detailed workings of SDMC with its behavior in various events like sender join, receiver join etc. In the next section III.4, we provide the evaluation of SDMC in various data-center network settings using the metrics like latency, network load variations, switch-memory utilization etc. and compare it with unicast and multicast for performance. In the section III.5, we discuss the previous works related to multicast in data center using SDN and their shortcomings. In the last section III.6, we conclude our work by providing the summary and directions for future work.

III.1 Motivation

III.1.1 Importance of Group Communication in Large Data Centers

Group communication is used when one participant (or multiple participants) needs to talk to multiple participants. Participant in this communication could be any entity like an application level abstraction, class object, process, host machine or IP address, person etc. Group communication is heavily used in the today's data center networks because various tasks that are routinely performed in data center networks inherently follow one-to-many or many-to-many semantics [20]. For example,

1. Elasticity: DCN use different elasticity techniques to provide commodity or tenant based services like PaaS, SaaS or IaaS. These techniques used for scaling up or down of computing, storage and network resources rely on group communication.

2. Fault Management: Fault management algorithms/strategies used by data centers to tolerate and mitigate faults require group communication at its core e.g. active and passive replication, fail-over, state synchronization in passive/semi-active replication, quorum management in active replication,

3. Access control/Privacy: Access control solutions like managing users, groups, passwords, user privileges etc. use group communication.

4. Security: Similarly security solutions like prevention, detection and removal of malware, virus etc. rely on group communication.

5. Application Management: DCN use application management tools for bulk software installment, software update, software upgrade, etc. All these need group communication.

All these tasks are routinely and heavily used in the data-centers and hence smallest overhead (network latency or load in this case) in them would be detrimental to the overall performance of data centers networks. Apart from above data center specific tasks, DCN also hosts many client applications which rely on group communication. For example,

1. Multi-player Gaming:

2. Multimedia Applications: video conferencing, on-demand video services like YouTube,

3. E-learning applications: Moodle, Coursera etc.

4. Collaboration Applications: online team editors like Google Docs, Microsoft 365,

5. Online storage Applications: e.g. Dropbox, Google drive, Microsoft OneDrive, Box etc.

6. Distributed Database applications: Hadoop etc.

Hence it is very obvious that efficient group communication is important for better performance of data center networks.

III.1.2 IP Multicast for group communication in DCN and its limitations

Traditionally IP multicast (IPMC) has been used (though quite minimally and ineffectively) for the group communication requirements of different applications. However, IPMC is very inefficient for adapting to dynamically changing network load and switch-memory. Due to these reasons, IPMC has not been seen large-scale deployment into the data center networks. Hence, dynamic and adaptive multicast protocol for the data center network is required to increase its use in the DCN.

III.1.3 SDN based multicast

In the recent years, new paradigm called Software Defined Networks has been emerged as a new way for managing networks. SDN decouples control plane of networking devices like (switches, routers, rate limiters, firewalls etc.) from its data plane. Due to very nature of SDN architecture, multicast has become easy to implement in the SDN controller as compared to the traditional IP multicast. However it too faces same challenges faced by IP multicast as described above. These challenges were very difficult to overcome in the legacy network where IPMC was implemented in the switches and was totally distributed. However softwarization of networking through SDN has made it possible to overcome these challenges. Also data center networks normally are composed of highly structured topologies and possess single window control of all the networking infrastructure as compared to wide-area-network or local area networks. Hence DCN provides great opportunity to use the SDN based multicast. In this work, we propose a novel way of using SDN based multicast(SDMC) for flexible, network-load aware, switch-memory efficient group communication specifically for the data center networks. SDMC efficiently uses combination

of unicast and software defined multicast and switches between them at run time agnostic to application and without any additional packet loss to find a better trade off to retain benefits of group communication while avoiding its disadvantages. In this work, we describe the design and implementation of SDMC using SDN controllers and OpenFlow enabled switches and evaluate it for various metrics like adaptiveness to network load and switch-memory utilization.

III.1.4 Our Contribution

In this work, we propose an new way of implementing multicast protocol for group communication protocol in SDN enabled networks specifically in large data centers networks. This multicast protocol will be more lightweight, dynamic, adaptive to networking resources like link utilization and switch memory when compared to the traditional multicast solutions. We list our contributions in this work as below,

- Design a network-load-adaptive and switch-memory-adaptive multicast for data center networks .
- Implement it as a SDN network application running on the top of SDN controller.
- Evaluate it for different data-center network load variations, switch-memory utilization.

III.2 Problem Description

IP Multicast

Communication between two or more participants can have different semantics based on number of participants on the sender side and receivers side.

- Unicast or one-to-one
- Broadcast or one-to-all

- Multicast or one-to-many or many-to-many
- Incast or many-to-one

IP multicast is standard for the multicast over UDP. IPMC assigns static multicast IDs (224.0.0.0 to 239.255.255.255) to the to be multicast group members (senders and receivers), creates routing tree before sender can send any payload data. There are three major ways by which IPMC generates routing trees. Common routing protocols used for multicast are as below.

- Internet Group Management Protocol (IGMP)
- Protocol Independent Multicast (PIM)
- Multicast BGP (MBGP)

IGMP is commonly used for IPv4 networks and MLD for IPv6 networks on the LAN(Local Area Network). PIM is used inside routing domain while MBGP is used between multiple routing domains.

We list the main reasons behind the lack of large scale deployment of the IPMC in the data center.

1. Static binding of multicast groups to sender and receivers.
2. Duplication of redundant multicast routing trees.
3. High overhead for creating multicast routing tree in the multi-sender scenarios.
4. Non-adaptive to the dynamically changing load.
5. Non-adaptive to the available switch memory.
6. Non-adaptive to the dynamically changing subscriptions or receivers.

In this section we will describe various requirements for designing a SDN based multicast solution (SDMC).

Flexible/Dynamic: Existing multicast protocols follow all or none semantics for multicast users. It allows users the choice of either using multicast for all the senders and receivers or not using it at all for all. It does not allow us to use selective multicast for few receivers or few senders while using unicast for remaining ones. SDMC should be capable of allowing applications to use multicast communication selectively as per application needs.

Initial Latency/Lazy initialization: In the existing multicast protocols like IPMC, creation/destruction of multicast senders/receivers immediately triggers creation/update/deletion of multicast routing trees. This incurs initial latency for the receivers especially for the highly dynamic subscriptionpublications and for the larger sized multicast groups. SDMC should be capable of reducing this initial multicast routing tree creation delay without compromising on the receiver performance. This is achieved by deferring the creation of multicast routing tree(lazy initialization) at later time when network and switch conditions are suitable for it.

Reuse of overlapping multicast routing trees: Existing multicast protocols makes it impossible to reuse partial or complete multicast routing trees due to flat nature of its multicast ids. This makes the scaling of multicast very difficult due to limited switch memory resources. SDMC should reuse the partially or completely overlapping multicast routing trees. For example if two (or more) multicast ids are having same (or almost) receivers (or switches to which receivers are connected), they should be able reuse the same multicast routing tree and hence save valuable switch memory.

Adaptive to network load: SDMC should be able to adapt to changing network traffic by switching between unicast and multicast. For example, if a (or more) link is under high load due to unicast traffic, SDMC should be able to switch that traffic to use multicast (if that traffic is part of group communication.)

Adaptive to switch-Memory: In data centers, switches come in various sizes and shapes. Depending on the specifications of the particular switch, switch memory, switching speed etc. may vary. Hence in a large data centers especially those that are in operation for long time, switches are heterogeneous. SDMC should be able to adapt to the switch-memory limitation scenario of the data center.

Consistent and Application-Agnostic SDMC : This consideration arises due to flexible and adaptive nature of new multicast protocol SDMC. Since, we are allowing multicast protocol to adaptively use either total multicast or partial multicast or total unicast dynamically based on the network load, switch memory or application requirements, a receiver may be changed from using unicast to multicast and to unicast during its life-cycle. Hence while switching between these configurations, SDMC is required to provide a consistent performance to all senders and receivers such that application remains unaware of these switching.

III.3 Solution Approach

In this section we will describe the implementation of the SDMC.

III.3.1 SDMC infrastructure

Figure 7 depicts the infrastructure required to implement the software defined networking based multicasting setup. It contains SDN enabled switches connected to form a SDN network with the control plane managed by the SDN controller (either centralized or distributed) and connected to a number of host machines (physical or virtual) with the SDN middleware (SDNMiddleware) installed on them.

Openflow enabled Switches: In a typical data-center network, number of openflow enabled switches are to be connected in a network using topologies like mesh, tree or jellyfish. Each such switch contains a OF-client to connect to the SDN controller. Also

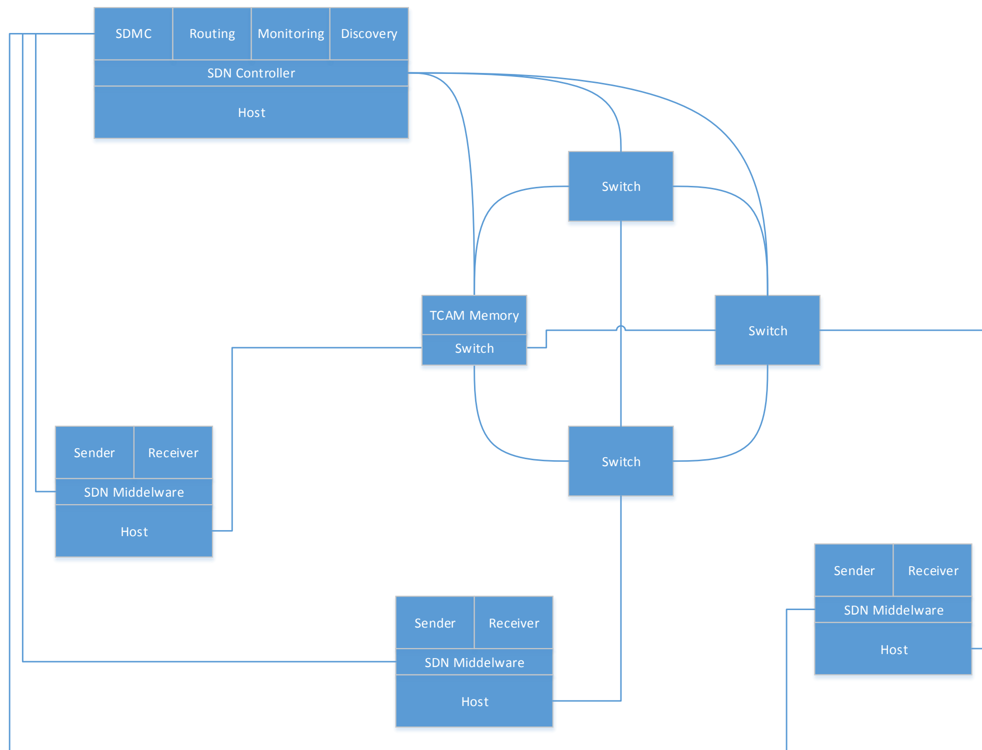

Figure 7: SDMC infrastructure

switch has fixed memory in the form of TCAM which is used to store the openflow rules for forwarding data packets.

SDN controller: Data center network is managed by the SDN controller. This SDN controller could be centralized or distributed. Though it is possible to use distributed controller, in this paper though we do not deal with the complexities arising from the distributed controller. We assume the SDN controller is on the dedicated machine with the dedicated connections to all the openflow enabled switches in the SDN network. For the SDN controller, we have various choices like NOX, POX, Floodlight, RYU, Opendaylight etc. SDN controller is part of the control plane in the SDN architecture as we discussed in the last section.

SDMC as SDN-NetApp: The core logic of Software Defined MultiCast(SDMC) is implemented as a SDN network application (SDN-NetApp). As described in the earlier section, network application forms the top layer of the SDN architecture and runs on the top of the SDN controller.

Other SDN Applications: As described above a major part of SDMC is implemented as a SDN network application(SDN-NetApp) on the top of the SDN control plane. However, we also need other SDN applications (apart from the SDMC itself) for the execution of the SDMC as described below. Some of these applications are general (like routing) but others (like host manager) are need to be specifically built for the SDMC type applications.

- Discovery: This SDN-NetApp provides service of automatic discovery of joining and leaving switches and hosts. It can assist SDMC for finding out joining-in or leaving-out recipients or senders of multicast group.

- Topology: This SDN-NetApp provides the service of topology creation out of the SDN network. It can be asked to create various topology among the switches and hosts. It does that by activating some links and de-activating others links.

- Monitoring: This SDN-NetApp is used to monitor and report various network properties like link bandwidth utilization, switch memory utilization etc.

- Routing: Routing SDN-NetApp is used to find (unicast) routes between two hosts using algorithms like OSPF. SDMC uses services of this SDN-NetApp to build a dynamic multicast routing tree at run-time.

- Network Virtualizer: If we are working in shared and multi-tenant SDN network then we need network virtualizer to slice the one physical SDN network space into multiple SDN networks. In this work though, we will deal with only non-virtualized SDN network. Hence we will not need Network Virtualizer.

- Host Manager: This SDN-NetApp is used to keep track of hosts connected to switches

and to communicate with them. This application is used by the SDMC to communicate with the SDN-Middleware of the host machines. This application is required as we are building a hybrid multicast protocol with the combination of application-level-multicast(or overlay multicast) and native network-level-multicast.

Host machines with SDN Middleware: Part of the SDMC which deals with application-level-multicast (or overlay multicast) is implemented with the help of a middleware (SDN Middleware) which runs on the top of the host machine connected to the SDN network. SDN-NetApp can control host network by communicating with this Middleware using the Host-Manager service application of SDN. This SDN middleware installed on the host machine is capable of things like translating a endpoint listening on a multicast id into multiple unicast ids, switching a endpoint from unicast to multicast or vice-versa without application intervention etc.

SDMC participants (Senders and receivers): The SDMC participants (senders and receivers) will run on the top of SDN-Middleware. SDN-Middleware will hide the various complexities arising out of dynamic, lazy and flexible SDMC from these participants. Sender and receivers will send/listen to a SDMC id and does not deal with (or know) whether underlying layers are using unicast or multicast or both.

III.3.2 Lazy Initialization

Initialization process of SDMC senders-receivers and SDMC routing tree creation is lazy to allow flexibility of adapting dynamically to the network load and switch memory limitations (described later in this section). SDNMCast exhibits this laziness in its workings while switching to multicast communication from default unicast. This laziness of SDMC can be seen in three different ways. First, when a new receiver requests to listen on SDMC-ID, it is not immediately added to the SDMC-ID as a multicast receiver but is added as an unicast destination on the all the existing senders of that SDMC-ID if any. Secondly SDMC multicast routing tree for new receiver is created in the controller but is not installed (in the

form of OF rules) in the switches immediately. And thirdly, when a receiver (or sender) leaves the SDMC-ID group, multicast tree is not updated immediately. All these above three decisions (viz. 1. when to add a receiver as a multicast destination 2. when to install multicast routing tree in switches 3. when to update the multicast routing tree after a receiver leaves) are taken by the SDMC holistically based on all other SDMC sender-receiver status and on the network load and switch-memory utilization instead of triggering them immediately. In the figure 8 and in figure 9, lazy initialization of the senders and receivers is described in the SDMC with the two-level SDMC-IDs.

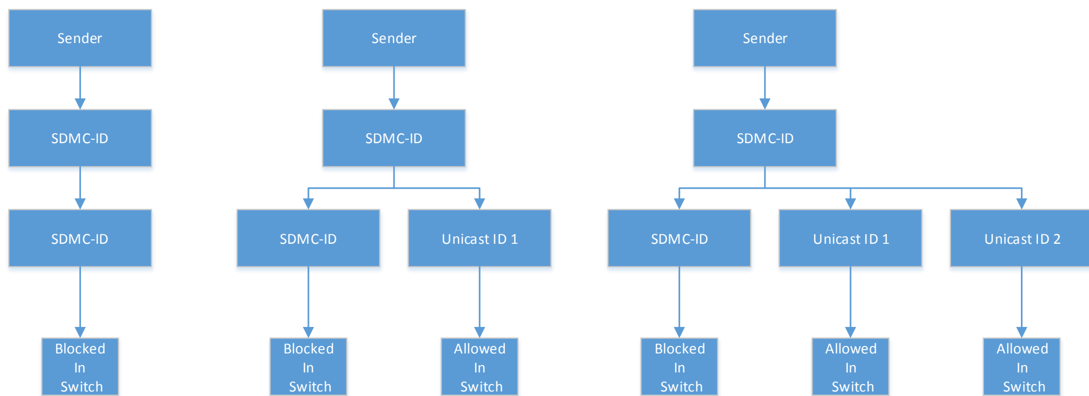

Figure 8: Initial SDMC Sender Setup

III.3.3 Two-level SDMC-ID

To allow reuse of the multicast routing trees, SDMC-ID space is divided into two viz. application-level (or external) and network-level (or internal). SDMC participants will deal with only external SDMC-IDs while network data-path will deal with internal SDMC-IDs. SDN-Middleware will be responsible for the translation of external SDMC-IDs to appropriate internal SDMC-IDs (and vice-versa). The SDMC (SDN-NetApp) will direct SDN-Middleware about use of appropriate translation and when to switch between different SDMC-IDs as described below. This two level SDMC-ID structure will allow SDMC to use

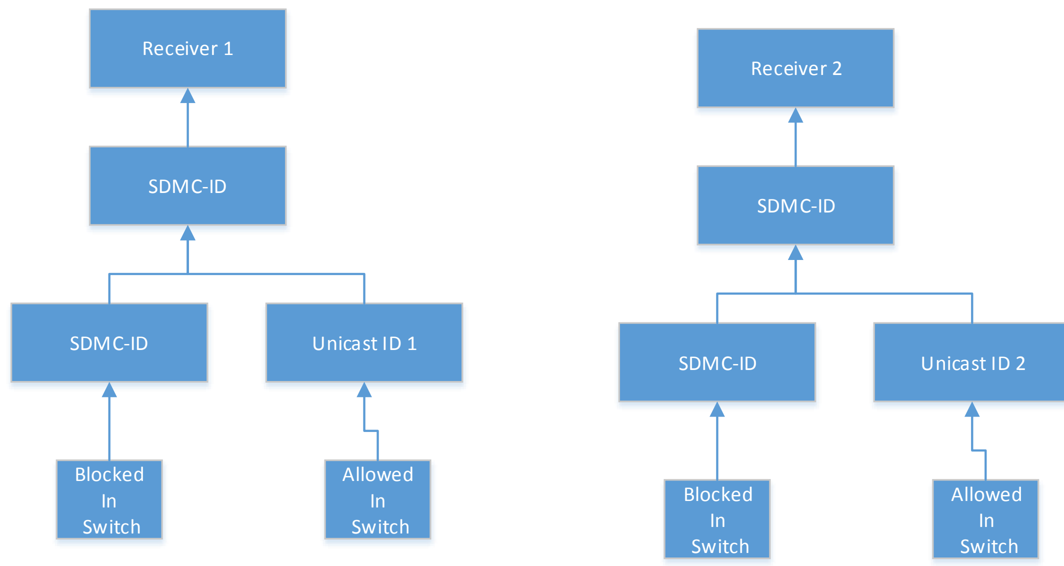

Figure 9: Initial SDMC Receiver Setup

same multicast routing tree (with the internal SDMC-ID) for the overlapping receivers of two different external SDMC-IDs. The decision about how to divide the address space into external and internal SDMC-IPs is left at the hand of the administrator of SDN network and can be configured at the network set-up time via the SDN controller configuration parameters.

III.3.4 Network link monitoring :

SDMC constantly keeps track of network links and their utilization with the help of monitoring network application. It then populates the link utilization information against the SDMC-IDs which are using that link for the unicast for a receiver as shown in the the Table 1.

III.3.5 Network Switch-memory monitoring:

SDMC also constantly keeps track of the memory utilization of the network switches with the help of the controller. Since controller installs rules in the switches, it knows

Table 1: Network Link Monitoring

Link	Switch No.	Switch No.	Utilization	SDMC-ID	Receiver (Unicast)
L1	Switch-03	Switch-05	17%	SDMC-ID-01 SDMC-ID-87	Rec-33 Recv-54
L2	Switch-20	Switch-11	32%	SDMC-ID-11 SDMC-ID-42 SDMC-ID-45	Rec-12 Recv-23 Recv-54
L3	Switch-31	Switch-21	31%	SDMC-ID-03 SDMC-ID-85	Rec-13 Recv-53
L4	Switch-13	Switch-03	69%	SDMC-ID-21 SDMC-ID-52 SDMC-ID-74 SDMC-ID-24	Rec-21 Recv-67 Recv-42 Recv-45
L5	Switch-3	Switch-9	81%	SDMC-ID-1	Rec-1
L6	Switch-15	Switch-21	70%	SDMC-ID-11 SDMC-ID-32	Rec-11 Recv-28

exactly how many OF rules are on each switches. Each switch comes with the maximum number of OF rules that it can accommodate. So we measure the switch-memory utilization as number of actual OF rules installed in the switch against the maximum number of OF rules allowed. For this work, we deal with OF 1.0 rules for all switches. However as newer version of OF standard are proposed, controller should keep track of different version of OF rule space for different switches. This table (Table 2) also keeps track of multicast receivers and associated SDMC ids for each switch.

III.3.6 Workings of SDMC:

We will now describe the SDMC with explaining sequence of activities executed by SDMC in response to various events like sender join, receiver join, sender leave, receive leave etc.

Sender Join: When a participant(sender) wants to send data on an application-level SDMC-ID, M^e,

Table 2: Network Switch Memory Monitoring

Switch No.	Available Memory	Used Memory	SDMC-ID	Receiver (Multicast)
Switch-00	10000	9132	SDMC-ID-01 SDMC-ID-19	Rec-65 Recv-65
Switch-10	20000	1313	SDMC-ID-41 SDMC-ID-64 SDMC-ID-08	Rec-43 Recv-84 Recv-63
Switch-32	20000	14434	SDMC-ID-13 SDMC-ID-43	Rec-15 Recv-14
Switch-12	30000	15151	SDMC-ID-01 SDMC-ID-18 SDMC-ID-34 SDMC-ID-75	Rec-64 Recv-24 Recv-47 Recv-51
Switch-32	10000	3234	SDMC-ID-14 SDMC-ID-65	Rec-76 Recv-54

- It sends a request to its SDN-Middleware. SDN-Middleware sends the request to the SDMC SDN-NetApp.
- SDMC SDN-NetApp assigns an appropriate internal SDMC-ID M^i to correspond to the requested application-level SDMC-ID, M^e.
- It also installs OF rule in the edge switch of the sender to block all the traffic with the destination id M^i.
- After than SDN-Middleware on the sender node installs a translation rule for $M^e <-> M^i$ in the host so that (1) when sender sends packets on external SDMC-ID M^e, it gets translated to internal SDMC-ID M^i and also (2) when any receiver receives the packet with SDMC-ID M^i, it gets translated to application level external SDMC-ID M_e.
- Additionally it also installs following translation rule $M^e -> (M^i, U_1, U_2)$ where U_1

and U_2 are the unicast destinations of the receivers of the application level multicast id M^e. This allows sender to start sending packets using unicast.

As seen from the above sequence of events, joining of a sender does not trigger creation or update of the multicast routing tree. This is possible because initially every sender is made to use unicast only. Later on as per the conditions of the network and the switch, sender are asked to switch between multicast and unicast. This is part of the lazy initialization process of SDMC.

Receiver Join: When a participant (receiver) wants to listen on an application-level SDMC-ID, M^e,

- It sends a request to its own SDN-Middleware which for-wards the request to the SDMC NetApp.
- SDMC NetApp retrieves the respective internal network-level SDMC-ID ,M^i, if available otherwise create new, and sends it to the receiver SDN-Middleware.
- SDMC then installs OF rule in the edge switch of the receiver to block all the traffic with the destination id M^i.
- SDN-Middleware on the receiver installs the two translation rules $M^e <-> M^i$ and $M^e <-> U^1$ where the U^1 is the unicast id of this receiver.
- SDN-Middleware also gives the preference to $M^e <-> U^1$ rule over $M^e <-> M^i$ so that first rule gets matched. This makes receiver to listen on unicast instead of multicast id. This is part of the lazy initialization of SDMC receivers.
- Meanwhile, SDMC-NetApp searches for all the senders of M^i and adds the unicast destination of U^1 in their SDN-Middleware translation rules.
- It then requests the unicast routing paths for this receiver to every sender of M^i from

the Routing-NetApp. Based on these routing paths, it then updates Table 1 and Table 2 with adding sender-receiver pair against each appropriate network link and switch.

Similar to joining of a sender, joining of a receiver also does not trigger creation or update of the multicast routing tree. This is possible because initially every sender is made to use unicast only. Later on as per the conditions of the network and the switch, sender are asked to switch between multicast and unicast. This is part of the lazy initialization process of SDMC.

Adapting to network load: As discussed above, whenever a link (or more) in the SDN network crosses the threshold load, the SDMC-NetApp is notified by the Monitoring-NetApp. This is done because SDMC-NetApp registers a listener event on the Monitoring-NetApp to notify it in the case of link crosses a particular threshold. Threshold is specified in the percentage of bandwidth utilization for a specific amount of time. For example 90% bandwidth utilization for a link for more than 30 consecutive seconds. SDMC-NetApp logic periodically (e.g. once in 60 seconds etc.) tries to reduce the network load by switching relevant unicast receivers to multicast receivers. It searches for the unicast receivers in the Table 1 against the link which is overloaded. SDMC-NetApp then switches these receivers from unicast to multicast either one by one or simultaneously. In the next subsection, we describe the process to switch from unicast to multicast without losing any packet or impacting the receiverś performance.

Switching from unicast to multicast: This event is triggered by the fluctuations in network link utilization as discussed above. Figure 10 shows the sequence of events that happen during a receiver is switched from unicast to multicast by the SDMC. So when SDMC wants to add receiver r1 to the multicast for a particular sender s1, it will execute following steps.

- SDMC will instruct the receiver r1 to listen on its unicast id U_1 along with multicast id M^i. At this point, receiver will not receive anything on its multicast id from sender

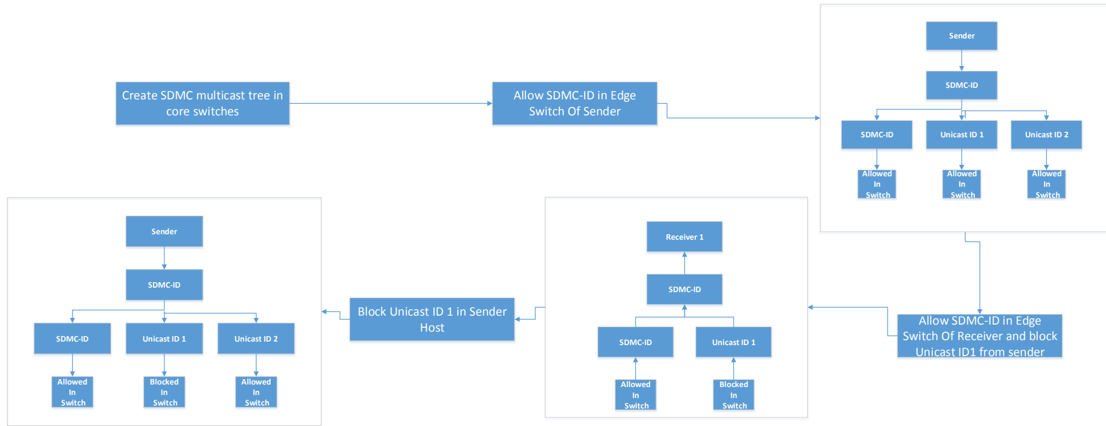

Figure 10: Adapting to Network Load

s1, since core switches and edge switches of sender/receiver are blocking the packets for multicast destination M^i.

- Then, it updates the multicast routing tree by adding appropriate OF rules to reach receiver r1 from the existing multicast routing tree of M^i in the core switches. At this point too, receiver, r1 will only receive packets on unicast id since edge switches of sender/receiver are blocking the packets for multicast destination M^i.

- Update multicast routing tree on the edge switch of sender s1 by adding/enabling OF rule for multicast ID M^i for receiver r1. (This last step is not required if one or more receiver, apart from r1, of sender s1 are using multicast and reached by same switch from the sender s1. This step is always required initially when there are no existing multicast receivers for the sender s1.) At this point too, receiver, r1 will only receive packets on unicast id since edge switches of receiver are blocking the packets for multicast destination M^i. However now there will be duplicate packets sent by the sender s1 in the network but without any receiver.

- SDMC now executes following two tasks atomically. (1) add a OF rule which blocks packets to the unicast destination of receiver r1 on the edge switch of sender s1. (2)

update multicast routing tree on the edge switch of receiver r1 by adding/enabling OF rule for multicast ID M^i to reach receiver r1.

- Atomicity of above two steps guarantees that no packet was lost and no packet was received more than once in the migration from unicast to multicast. This however does not prevent senders from sending packets on both unicast and multicast IDs during the time step 3 is started till the time last step in this sequence is finished.

- Later, when receiver starts to receive packets on its multicast id, it will stop listening on its unicast id.

- Receiver then notifies SDMC that it is now listening on only multicast ID, SDMC will then update a translation rule in the SDNMiddleware of s1 by removing the unicast address of receiver r1 (u1) from its mapping. i.e. $M^e- > (M^i, U_1, U_2)$ becomes $M^e- > M^i, U_2$. At this point, sender will stop sending packets to unicast destination of receiver r1.

However for efficiency, SDMC should not switch single receivers-sender pair from unicast to multicast but perform bulk switching periodically.

Adapting to switch-memory utilization: Similar to network load monitoring, Monitoring-NetApp monitors the network switch memory utilization too. Hence, whenever a memory utilization of a SDN switch in the SDN network crosses the threshold limit, the SDMC-NetApp is notified by the Monitoring-NetApp. This is done after SDMC-NetApp registers a listener event on the Monitoring-NetApp to notify it in the case of switch memory utilization crosses a particular threshold. This Threshold is specified either in the percentage of memory utilization of a switch or number of OF-rules installed on the switch for the SDMC. In this work, we take the later approach of counting the switch-memory in the form on number of OF-rules. The reasoning behind this approach is that it specifically measures the switch-memory utilized for the SDMC and not other SDN-network applications like unicast routing or load balancing etc. SDMC-NetApp logic periodically (e.g.

once in 60 seconds) tries to decrease the memory utilization of the switch by switching relevant multicast receivers to use unicast communication. To do that, it searches for the multicast receivers in the Table 2 against the overloaded switch. . SDMC-NetApp then switches these receivers from multicast to unicast either one by one or simultaneously. In the next subsection, we describe the process to switch from multicast to unicast without losing any packet or impacting the receiverś performance.

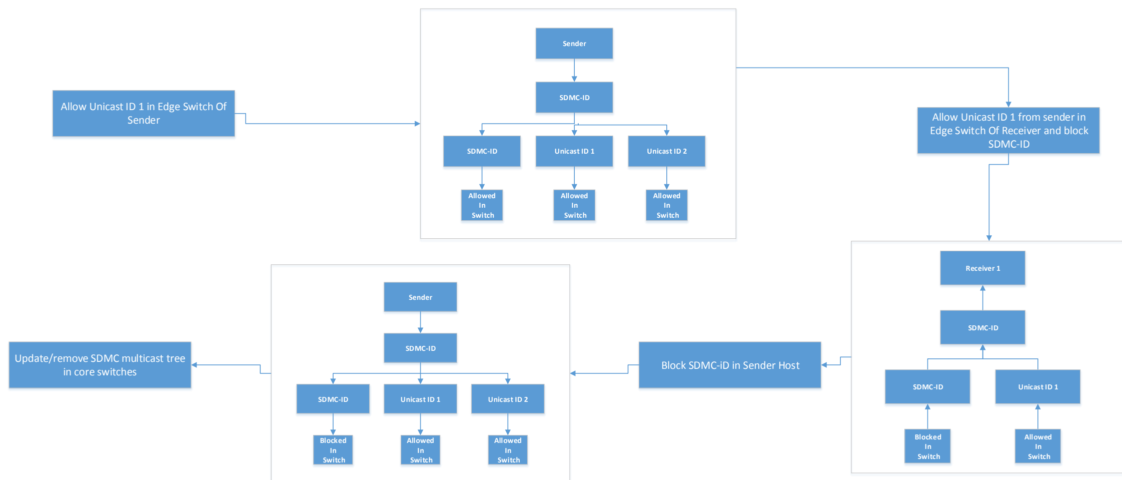

Figure 11: Adapting to Switch Memory Utilization

Switching from multicast to unicast: This event is triggered by the changes in switch memory utilization as discussed above. Figure 11 shows the sequence of events that happen during receiver is switched from multicast to unicast by the SDMC. So, when SDMC wants to remove receiver r1 from the multicast for a particular sender s1,

- SDMC will instruct the receiver r1 to listen on its unicast id U_1 along with multicast id M^i. At this point, receiver will not receive anything on its unicast id from sender s1, since core switches and edge switches of sender/receiver are blocking the packets for multicast destination M^i and sender is not sending any packet on the unicast id of the receiver.

- SDMC will then update a translation rule in the SDN-Middleware of sender s1 by adding the unicast address of receiver r1 (u1) in its mapping. i.e. $M^e - > (M^i, U_2)$. becomes $M^e - > M^i, U_1, U_2$. At this point, sender will start to send on both the unicast and multicast. But at this point too, receiver will not receive anything on its unicast id from sender s1, since core switches and edge switches of sender/receiver are blocking the packets for multicast destination M^i.

- Now, SDMC updates the multicast routing tree for multicast id M^i, on the core switches by disabling/removing OF rules which to reach receiver r1 for multicast id. At this point too, receiver will not receive anything on its unicast id from sender s1, since edge switches of sender/receiver are blocking the packets for multicast destination M^i.

- SDMC now executes following two tasks atomically. (1) remove the OF rule on the edge switch of sender s1 which blocks packets to the unicast destination of receiver r1. (2) update multicast routing tree on the edge switch of sender s1 by disabling/removing OF rule to reach receiver r1.

- Atomicity of above step guarantees that no packet was lost and no packet was received more than once in the migration from multicast to unicast. This however does not prevent senders from sending packets on both unicast and multicast IDs during the time step 3 is started till the time previous step in this sequence is finished.

- Later, when receiver starts to receive packets on its unicast id, it will stop listening on its multicast id.

- Receiver then notifies SDMC that it is now listening on only unicast ID, SDMC will then update the multicast routing tree on the edge switch of the receiver r1 such that packets for destination M^i will not reach receiver r1.

However for efficiency, SDMC should not switch single receivers-sender pair from multicast to unicast but perform bulk switching periodically.

Receiver Leave: When a receiver wants to leave the multicast group, its SDN-Middleware notifies the SDMC with "Receiver_Leave" notification. SDMC first checks if the receiver is using unicast or multicast. If receive is on unicast mode then SDMC only need to remove it from the translation rule of its senders. However if receiver is using multicast then SDMC needs to update the routing tree. However SDMC does not need to do this action immediately. But it only removes that OF rule from the edge-switch of receiver which for-wards dest=M^i packets to this receivers. Remaining multicast routing tree is cleaned up during the next periodical switch-memory monitoring event. So if same receiver (or another receiver connected to same switch or another receiver which can be reached via the same switch) joins again (before cleanup), SDMC takes lesser time in updating the multicast routing tree for it.

Sender Leave: When a sender receiver wants to leave the multicast group, its SDN-Middleware notifies the SDMC with "Sender_Leave" notification. SDMC then add OF rule in the edge switch of the sender to block the traffic from the sender s1. At this point, there will be no traffic from sender in the network. Then SDMC asks SDN-Middleware of sender to delete the sender. SDMC defers the removal of multicast routing tree of the M^i used by sender to later time.

III.4 Experimental Evaluation

In this section, we measure the performance of the SDMC against unicast and traditional multicast using IPMC for various performance metrics like average latency for receivers, switch-memory utilization efficiency, network load aware and packet loss. As we described in the design considerations, SDMC is hybrid and flexible in a sense that it tries to combine the benefits of both unicast and multicast at run time. Hence in this section, we

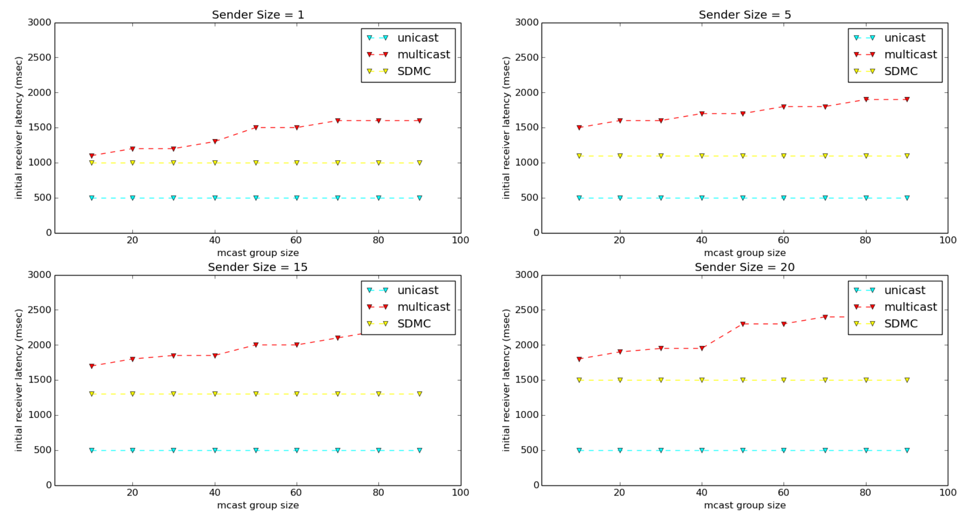

Figure 12: Receiver Latency against group size

will compare SDMC with the unicast and multicast (IPMC) and see how overall SDMC performs better than both the approaches.

III.4.1 Average latency for receivers:

We will measure the performance of our approach against the generic IPMC in terms of the latency for the receivers. As the size of multicast group becomes more dynamic i.e. more number of receivers/senders join or leave the multicast group per second, in the traditional multicast time to update the multicast routing tree increases which is mandatory before new receiver starts to receive any packets. However in SDMC, since SDMC can switch to unicast and take benefit of 0 latency during this time, overall SDMC performs better. Figure 12 shows initial receiver latency comparison as senders increases for a fixed topology (jellyfish) and fixed network size (10 networks switches). Figure 12 shows receiver latency comparison as network size(in terms of switches) increases for a fixed mcast group size (10) and fixed number of senders (1). In both the cases SDMC can be seen scaling better in terms of receiver latency as compared to basic multicast.

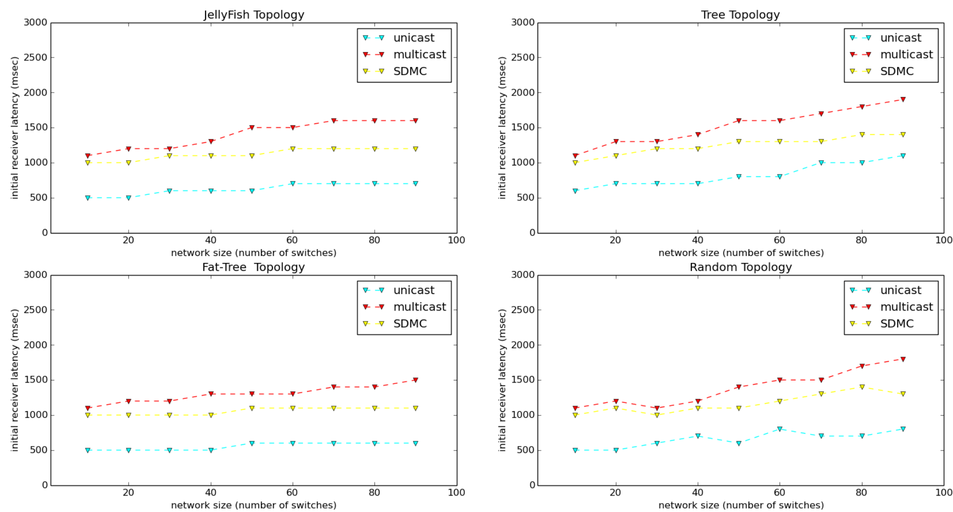

Figure 13: Receiver Latency for different network topology and size

III.4.2 Network Load Adaptive-ness:

We will measure the performance of our approach against dynamic network load. Basic unicast (when used in one-to-many communication) increases the network load since it has to duplicate same packets for every receivers. The traditional multicast (IPMC) reduces the network load compared to basic unicast. However it increases latency for receivers and (as we will see in the next section) also increases switch-memory utilization. SMDC efficiently switches between unicast and multicast based on network load. When network load is less, SDMC uses unicast while it switches to multicast when network load increase. Hence as seen from figure 14, SDMC adopts to network load by switching between unicast and multicast.

III.4.3 Switch-Memory Utilization Adaptive-ness:

We will measure the performance of our approach against dynamically changing switch memory utilization. The traditional multicast (IPMC) requires large amount of switch-memory compared to basic unicast. However it reduces network load as we saw before. SMDC efficiently switches between unicast and multicast based on switch-memory. When

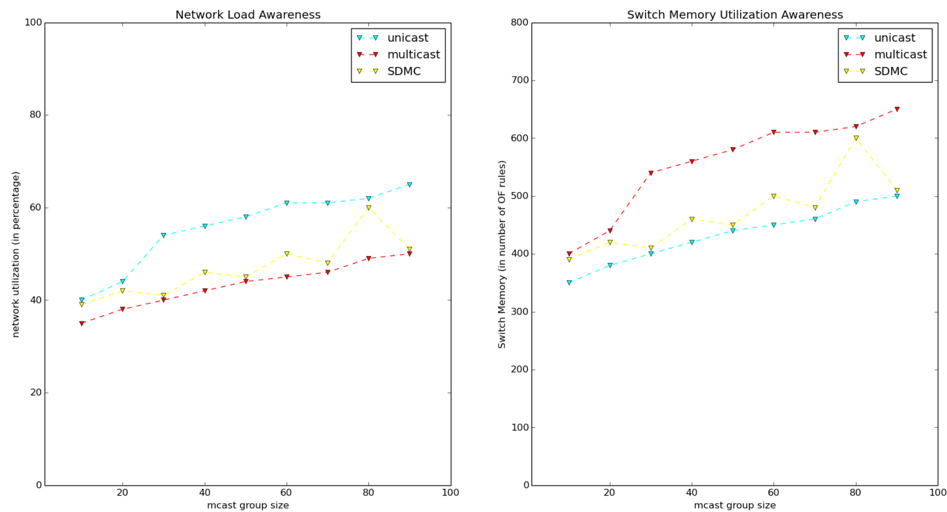

Figure 14: SDMC adaptive-ness to network load and to switch memory utilization

switches have sufficient remaining memory, SDMC uses multicast but when when available switch memory becomes less, SDMC starts to use unicast. Hence as seen from figure 14, SDMC adopts to switch-memory and performs better than multicast.

III.4.4 Packet Loss

Now, we will measure the packet loss during SDMC as compared to basic unicast and multicast. As seen in the figure 15, SDMC does not decreases the packet loss as compared to basic multicast (IPMC). The reason behind this result is that SDMC switches to unicast when network is not heavily loaded. Hence during this time SDMC suffers less packet loss as compared to IPMC hence overall SDMC performs better than basic IPMC.

III.5 Related Works

In this section we will describe some of previous works related to multicast in data center networks. These approaches do not use SDN specifically for improving multicast problems but use other ways like rich path diversity or steiner tree etc. In [21], authors describe a way to improve multicast in data center networks by using rich paths available

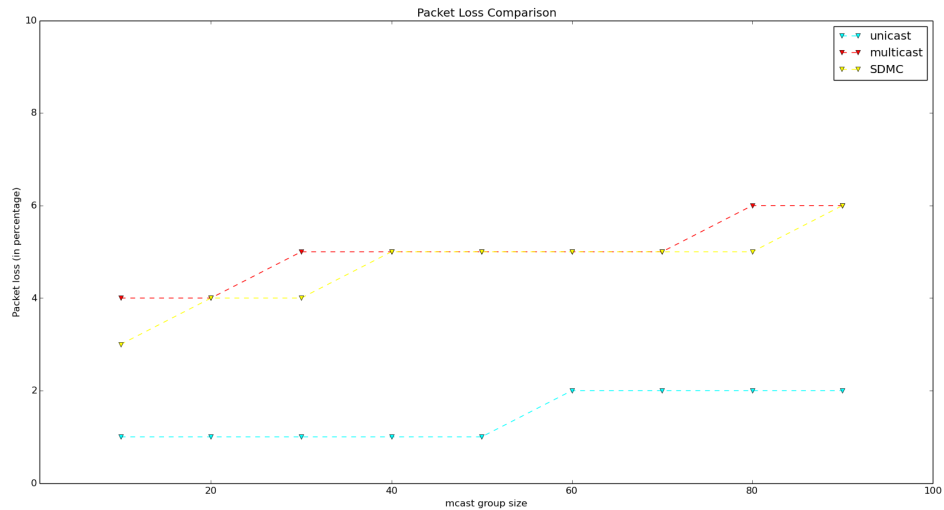

Figure 15: SDMC Packet loss

in large and highly connected data center networks. This approach creates backup overlays for every multicast routing tree as switches to it as per network load fluctuations. In [20], authors describes a technique to improve the multicast latency by compressing the tree using a multiclass bloom filters. In the same way, authors in [15] presents a optimization multicast routing tree creation by using steiner tree approach.

Now, we will describe some SDN specific multicast approaches. Authors in [38] proposes a OpenFlow based multicast approach for efficient SDN based multicast. Authors in [16], presents an SDN enabled technique for implementing multicast in large data centers networks called Avalanche. Avalanche creates bandwidth efficient multicast routing trees using its algorithm called as AvRA which efficiently uses the topology information of data center networks. It also uses global visibility and centralized control provided by centralized SDN controller. In [13], authors describe a SDN enabled efficient multicast scheme especially for IP-over-OBS networks. In [23], authors propose another SDN enabled multicast scheme which uses the knowledge of anticipated processing time for each route based on history data to use the optimal routing tree. However all the above SDN

based approaches suffer from same issues suffered by basic multicast e.g. scalability and latency since it does not adopts to changing network load and switch-memory.

III.6 Concluding Remarks

Data center networks heavily rely on the use group communication for various tasks. Data center management utilities (e.g. software update/upgrade, log management, resource monitoring, scaling various resources up or down, access control etc.), collaborative applications like social media, project management tools, version control systems etc.), multimedia applications, multi-player games are few examples of tasks that require efficient group communication. However though multicast is useful for efficient group communication, IP multicast has seen very low deployment in the data center networks due to its deficiencies like inefficient scaling, inefficient switch-memory utilization, initial receiver latency. With the advent of SDN (Software defined Network), though, multicast has become easy to implement in the SDN controller, it too faces challenges faced by IP multicast. In this work, we propose a novel way of using SDN based multicast(SDMC) for flexible, network-load aware, switch-memory efficient group communication specifically for the data center networks. SDMC efficiently uses combination of unicast and software defined multicast and switches between them at run time agnostic to application and without any additional packet loss to find a better trade off to retain benefits of group communication while avoiding its disadvantages. In this work, we have described the design and implementation of SDMC using SDN controllers and OpenFlow enabled switches and evaluate it for various metrics like initial receiver latency, network load awareness, switch-memory utilization efficiency.

CHAPTER IV

SOFTWARE DEFINED WIRELESS MESH NETWORKS

Software Defined Networking (SDN) has seen growing deployment in the large wired data center networks due to its advantages like better network manageability and higher-level abstractions. SDN however has been slow to be used in the wireless scenario like wireless mesh networks (WSN). This is due to the fact that SDN (and its underlying Open-Flow protocol) was designed initially to run in the wired network where SDN controller has wired access to all the switches in the network. Various workarounds have been proposed for adapting SDN and Openflow to the wireless setting. However all these approaches require some kind of hybrid switching hardware and software (especially for routing) which goes against the fundamental SDN architecture and also causes unnecessary increase in hardware and software complexity of the switch. To address this challenge, we propose a pure opneflow based approach for adapting SDN in wireless mesh netowrks by extending current OpenFlow protocol for routing in the wireless network. We describe the extension to OpenFlow protocol and also its use in a novel three stage routing strategy which allows us to adapt a centralized routing of SDN in an inherently distributed wireless mesh network without requiring additional support from switch hardware. We evaluate our approach with the existing hybrid approach using latency metric for controller-switch and switch-switch connections.

IV.1 Introduction

IV.1.1 Wireless Mesh Networks (WMN)

Wireless mesh network (WMN)[2, 22] is a special kind of mobile ad-hoc network which consists of mobile end hosts and wireless routers. Each end host is connected to the network via one (or more) wireless routers over radio using peer to peer (P2P) connection.

Mesh is connected to the internet (or outside world) through one or more routers called as gateway routers using either cellular network link or wired link. Other routers obtain the internet connection via one of the gateway routers. Both routers and end hosts are mobile nodes. However routers are supposed to be less mobile as compared to end hosts. The main difference between wireless mesh and other wireless networks like WLAN (Wireless local area network), WMAN (wireless metropolitan area network), MANET (Mobile ad-hoc network), WSN (wireless sensor network)[6] is the role played by mesh routers in WMN. Mesh routers in WMN work rather independently but can also collaborate with each other to form a larger mesh dynamically. This has made mesh network preferred way of network architecture for internet of things (IoT) network scenarios in domains like intelligent transportation and industrial automation [34].

IV.1.2 Software Defined Wireless Mesh Networks (SD-WMN)

In the recent years, research is going on to adapt SDN to wireless mesh network setting[10, 14, 27, 34, 35]. As mentioned earlier SDN basically envisions network having homogeneous, static and centrally controlled network topology. But wireless mesh is the exact opposite of it and consists of highly dynamic and distributed network topology. This creates multiples issues for using SDN architecture for wireless mesh network. In traditional wireless mesh network, router (generally referred to as a switch in SDN terminology) communicates with its neighbors to arrive at routing paths among themselves using Ad-Hoc On-Demand Distance Vector Routing (AODV) or Optimized Link State Routing protocol (OLSR). In SDN though, routing is exclusively controllerâĂŹs job. Controller, however may not know whole topology for making routing decisions since it may not be directly connected to all the switches in the first place[29]. Hence before controller starts any routing algorithm, it has to establish connection to all the switches. This makes routing in SDN based wireless mesh network more complex than that in a traditional mesh networks. Previously, researchers have proposed hybrid approach for converting wireless

mesh into SDN. Such approach requires each switch to be equipped with both the technologies (SDN-Openflow and legacy routing protocols). This increases the hardware and software complexity of the switches and also does not provide 100% pure software based wireless mesh network as some part of the network still functions in non-SDN way. We will discuss this hybrid approach in detail in the chapter IV.2.

IV.1.3 Contributions & Outline

In this work we provide a better approach for converting wireless mesh networks into purely software defined networks. To achieve this we propose a novel way of performing routing in three stages in the SDN based wireless mesh network by using modified OpenFlow protocol which allows us to remain faithful to SDN philosophy of keeping switch design simpler and of centralized control plane while also allowing flexibility & mobility inherent in the distributed wireless mesh network. We propose three stage routing which consists of following stages.

1. Initial flooding based distributed non-optimized routing between controller and switch

2. Centralized optimized shortest path routing between controller and switch

3. Centralized shortest path routing among switches

This chapter is organized as below. First we describe the differences between non-SDN based wireless mesh network and SDN based one in chapter IV.2. Chapter IV.2 then discusses motivation behind the design and architecture of our routing strategy. Next in chapter IV.3, we describe each stage of routing in details. We describe the implementation followed by the evaluation in next chapter IV.4. In the end we conclude with discussion about possible future direction in chapter IV.5.

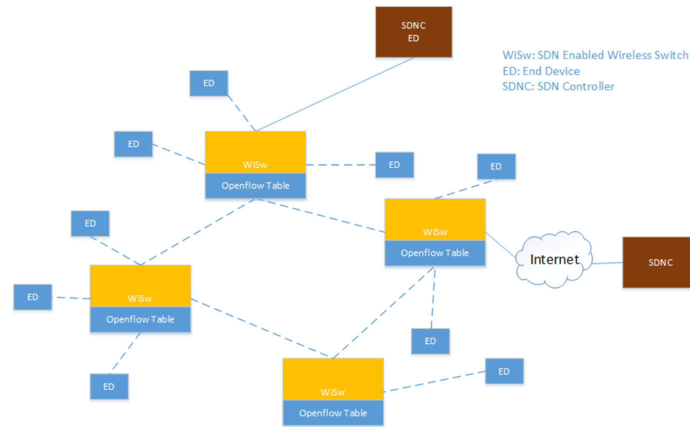

Figure 16: SDN Based Wireless Mesh Network

IV.2 Desgin

Figure 16 describes a general use case scenario of SDN based wireless mesh network. Openflow-enabled wireless switches are distributed across a geographical field. These switches could be either stationary or mobile. One or more of these switches are connected to the internet. This connection could be either wired or cellular. SDN controller is connected to one or more of these switches either directly by a wired connection or over internet. Mobile end devices are connected to switches via access points (not shown in the figure). Wireless switches are white-box SDN-enabled (Openflow enabled) switches. They do not contain any software apart from an openflow client and flow table. Basic operations of these switches are to receive packet, match up against Openflow rule and take appropriate action. Architecture of wireless mesh network allows various types of mobility and dynamism in it. End device can move. Wireless switch can move along with its access points.

IV.2.1 Motivation behind Three-Stage Routing

In traditional wireless mesh networks, nodes (mesh routers) communicate with each other using routing protocols like AODV and OLSR. This is inherently distributed routing. However the goal of SDN is to centralize the network control as much as possible. In

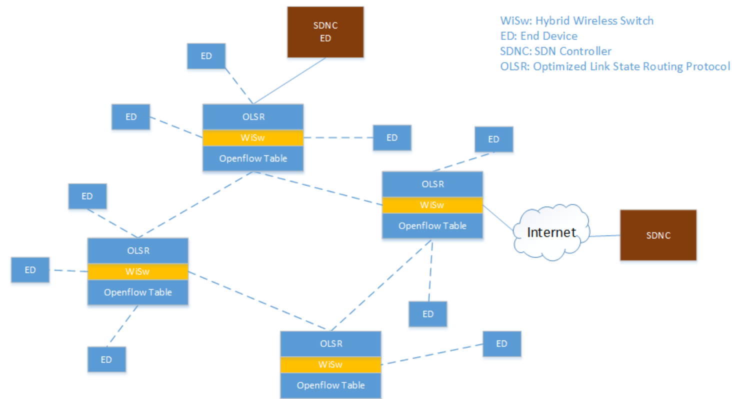

Figure 17: Hybrid Architecture for SDN Wireless Mesh Network (1)

traditional wireless mesh routing is done using either AODV or OLSR where each node (wireless switch) will broadcast information about its directly connected end devices to all other nodes. Using this information each node will derive its own routing path to all the other nodes independently and in a distributed fashion.

There are major challenges in using routing algorithms in architecture in figure 16 as compared to traditional non-SDN wireless mesh network because of following differences in architecture. In SDN, before deciding routing among switches, switch needs to establish a connection to the controller. In SDN, switch only supports basic forwarding capability with Openflow client. Switch does not support any complex software hence cannot use highly distributed algorithm like AODV or OLSR. Switch needs control instructions from controller to perform any intelligent action. Another major difference between SDN and non-SDN wireless mesh network is that in SDN control decisions are taken by the centralized controller. However in traditional wireless mesh network, control decisions are taken in a distributed manner by the algorithms installed in every switch which themselves are distributed. Such distributed architecture helps to allow mobility in the network. SDN being centralized does not inherently support mobility of its network nodes (hosts, switches, controller etc.) However we need to support such mobility if we want to extend SDN to the wireless mesh networks.

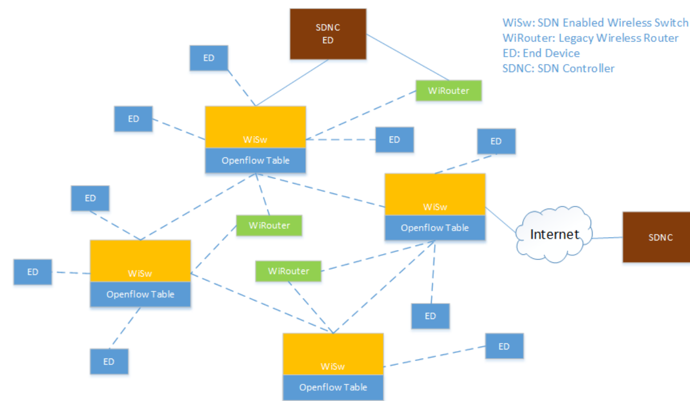

Figure 18: Hybrid Architecture for SDN Wireless Mesh Network (2)

IV.2.2 Existing Solutions

Wm-SDN[8] tries to reconcile this problem by using hybrid protocol. It uses traditional distributed protocol AODV for switch-controller connection. And when controller-switch connection is established, it then used SDN based centralized protocols for switch-switch routing decisions. This architecture is shown in the figure 17 where each switch requires to support both SDN Openflow and also legacy routing protocols. Hence this approach makes network not 100% software defined since switch is involved during first part of routing decisions (i.e. during AODV). Also it requires switch to support complex hardware and software than SDN envisions.

Other works [1, 25] have also proposed hybrid approach for routing though of different kind. They propose to combine SDN enabled switches and legacy switches (or routers) in a wireless mesh router as shown in the figure 18 where SDN enabled switches form the SDN network while traditional switches form the legacy network. It makes this approach also hybrid one. In this architecture, legacy switches run traditional routing algorithm (OSPF in this case) while every SDN enable switch has to be in direct contact (wireless or wired) with one of such legacy switch. This allows SDN-enabled switch to not support any complex hardware and software. However it requires each SDN-enabled switch to communicate with at least one legacy switch.

IV.2.3 Design considerations

In the view of above discussion about differences in two architectures and the motivation behind this work, we list the design considerations or requirements that we try to meet while designing architecture of the SDN based wireless mesh network routing.

1. Design of wireless switches hardware should remain simple i.e. switch should only support SDN Openflow protocol.

2. Switch should not require to install any non-SDN specific software.

3. Architecture should allow mobility of its nodes.

IV.3 Three-Stage Routing: Architecture

To work under above constraints and requirements, we propose pure openflow based three level routing strategy for SDN based wireless mesh network as described below.

Stage 1: Initial Controller-Switch Connection

As described in figure IV.3, in the SDN based wireless mesh networks, only few switches are directly connected to controller. So first task is to connect all the switches to (at least one) controller by setting up initial/basic routing. We propose an initial (non-permanent) routing stage where controller will find all the switches through flooding the network without considering whether the path it finds is best or not. To achieve this, we use an Openflow based routing algorithm for initial controller-switch connection by adapting OLSR in two ways. First, instead of switches broadcasting their link state (i.e. information about directly connected end devices), controller will broadcast information about its directly connected switch. Second, instead of running full-fledge wireless mesh routing protocol like AODV or OLSR in switches, we modify Openflow client in switch such that switch finds the initial path to the controller without requiring any additional software.

Stage 2: Controller-switch path optimization

Once initial connection is established, the routing paths set up in this stage will be used

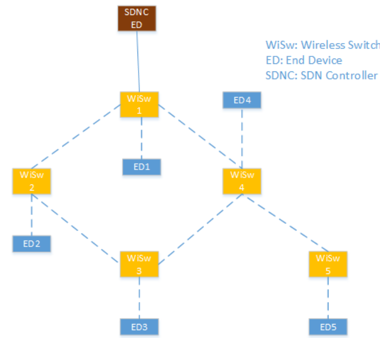

Figure 19: Example SDN Mesh Scenario

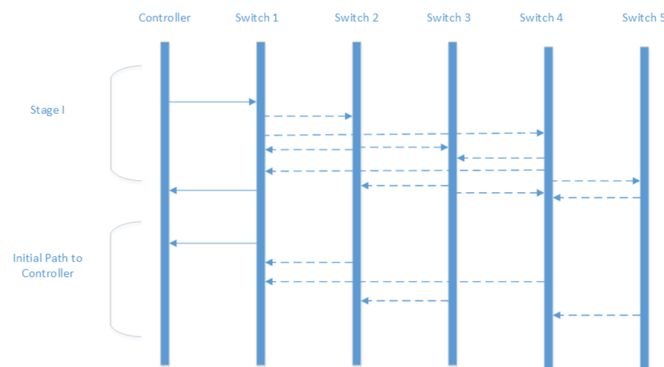

Figure 20: Stage I Interactions

do install new alternative (shortest, optimum or load balanced) paths. Controller can decide these alternative paths between itself and a switch since at this stage controller has the global view of network. Then it will install them in switches using original non-optimized paths.

Stage 3: Routing Among Switches

After second step, controller will derive the shortest path routing among switches themselves and it will install these routing paths in them via the shortest paths set-up in the previous stage.

As described above, to achieve above routing strategy, we needed to modify the Openflow client. Next, we will describe modifications that we make to the Openflow client by introducing three new message types and semantics to achieve this.

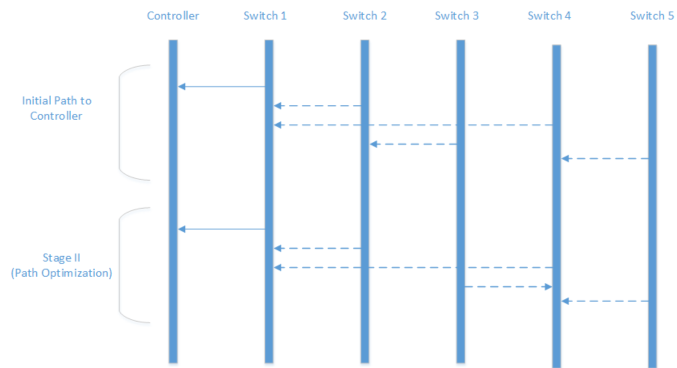

Figure 21: Stage II Interactions

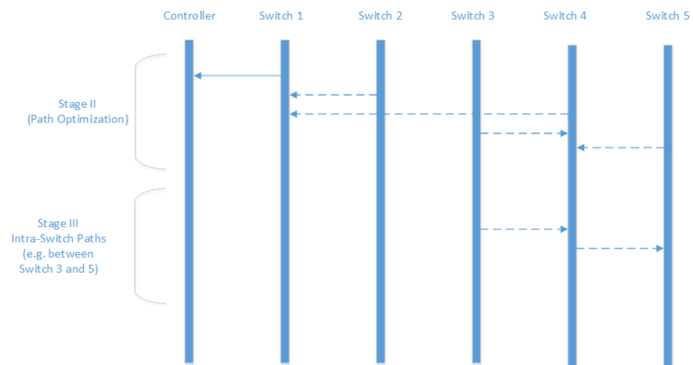

Figure 22: Stage III Interactions

IV.3.1 OpenFlow Modifications for staged routing

Here we describe modifications to Openflow protocol with proposed new message types which will be used to establish an initial connection between wireless switch and controller. These messages allow wireless switch to establish an initial, non-optimized connection to the controller without using resource-heavy wireless mesh protocols like AODV and OLSR. Once a switch establishes initial multi-hop connection to controller, routing for this connection can be optimized using the centralized routing information present in the controller.

1. *OF_Initial_Path_Request:* Initially controller will send this OF message to all its directly connected switches. As described in the architecture figure IV.3, switch could be either connected via wired interface or wireless interface to the controller. Also controller could be at the same location as that of switch or could be situated in the cloud data center. Once a switch receives this messages, it updates the controller path destination with the source id found in the received OF_Initial_Path_Request message. Switch then creates new OF_Initial_Path_Request message with its own source id and broadcasts it to the other switches. This step is performed periodically (e.g. every 30 seconds) by every switch. Period of this can be set-up statically or can be adjusted based on the dynamic properties of the network. In this way, every switch that receives OF_Initial_Path_Request message establishes an initial path to the controller. This path may not be the shortest but only be used as a first step for obtaining the shortest path in the next stage. Also as every switch broadcasts this message periodically, this helps to handle the mobility in the network.

2. *OF_Initial_Path_Response:* Switch sends this message to the controller on the initial path found in the stage I. This message is directed towards the controller and is only sent to that neighbor from which switch received OF_Initial_Path_Request message

first i.e. this messages is sent via initial path between switch and controller. However, this response message contains SSIDs of all the neighboring switches i.e. all the neighboring switches from whom this switch received OF_Initial_Path_Request message.

3. *OF_Controller_Shortest_Path:* This openflow message is used to optimize the initial connection path between controller and switch. Controller sends this message to switches to update path to controller with the shortest path. This message is sent only when the initial path differs from the shortest path between controller and switch. Also this message is always sent using the initial path. When switch receives this message, it installs new path to controller which is shorter than previous path. And switch gives this new path higher preference by installing rule for this path before the rule of initial path. So from now, whenever switch sends any message to controller, it takes the shortest path. Only when shortest path fails to deliver message, initial path is used.

We will describe message flow in three stage routing approach using use case in figure 19. Figure 19 shows a SDN based wireless mesh network with five wireless switches. Each switch is connected to a single edge device while only first switch is connected to the controller directly. Figure 20 describes the stage-I interactions. In stage-I, controller sends OF_Initial_Path_Request to switch-1. Switch-1 then duplicates this message and sends to switch-2 and switch-4. This allows each switch to find a initial path to the controller. Figure 21 describes the stage-II interactions. Each switch sends OF_Initial_Path_Response message to the controller via the initial path found in the stage-I. This message contains information about neighboring switches. In our example, switch-3 has received OF_Initial_Path_Request from two switches (Switch-2 and switch-4). However it received message from switch-2 first. Hence for switch-3, initial path to controller is via switch-2. However, when switch-3 replies to controller using OF_Initial_Path_Response message via initial path (i.e. via switch-2), it includes ssid of switch-2 and switch-4 in it. This

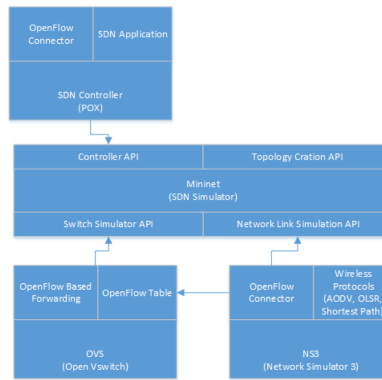

Figure 23: Implementation components

allows controller to deduce the neighbors of each switch and hence the topology of the network. Using knowledge about topology of the network, controller now installs the shortest path routes in switches as shown in the figure 21 where switch-3 now has a shorter path to controller via switch-4 instead of via switch-2. Finally in the stage-III (figure 22), controller installs routing paths among switches using the shortest path from stage-II. As shown in figure 22, controller installs Openflow rules such that switch-3 can reach switch-5 via switch-4.

IV.4 Experimental Evaluation

We have implemented two level routing strategy using SDN based emulation framework Mininet. Basic Mininet does not provide the wireless link support. Mininet-Wifi provides basic support for simulating wireless links but lacks support for essential wireless network based algorithms like shortest path or AODV or OLSR which are normally supported by other network simulator like ns2 or ns3. Hence, for this we used ns3 which is network simulator with support for wired and wireless links. NS3 provides support for Openflow client which is required for SDN based switch. We have used OpenNet to interface ns3 Opneflow client with the Mininet simulator. Opennet simulator combines ns3 with Mininet to provide wireless simulation in the SDN based network settings.

As shown in the figure 23, Mininet simulates switch using Open Virtual Switch (OVS).

Openflow client of OVS is interfaced with the NS3 Openflow connector using the OpenNet APIs. This allows Mininet to use wireless support for the links between switches. NS3 also provides off-the-shelf support for various wireless mesh network algorithms. Mininet on the other hand creates software switches. For this work we have used OVS switch. However Mininet has support for different types of software switches. Mininet also helps to creates various types of topologies in the wired network like tree, fat-tree, jellyfish etc. However since we are using wireless mesh network, such topology is not useful. We need support for mobile nodes (switches and hosts). This is done through NS3 simulator. However since currently there is no direct interface between NS3 and Mininet for creating topology of mobile nodes, we hardcoded topology in the NS3 and then migrated it to Mininet in offline fashion. However in the future we plan to create NS3-Mininet AOI for wireless mobile topology generation. Once the wireless mesh network (of mobile switches and mobile host) is created using Mininet and NS3, SDN controller is connected to one or more of the wireless switches. For this purpose we have used POX controller. As shown in the architecture diagram(figure 16), controller is directly connected to one or more of switches or is placed in the cloud. Currently we are not considering later case. In our implantation, POX controller is directly connected to one of the switches in the Mininet in hard-coded fashion. In the mininet, when we add a controller in the network, by default every switch is connected to it. This is useful because in wired data center, it is default case. However in wireless mesh network, we need to make sure that controller is connected to only few of the switches. In current Mininet this is not possible without modifying Mininet source code. We have found workaround to this problem by installing dummy or proxy rules in those switches which are not supposed to be connected to controller directly. These rules basically direct the switch to a non-existent controller (instead of existing controller). As result of this, switch thinks it is not connected to any controller. Three stage routing strategy is implemented on the top of POX controller as a network application. This strategy also makes use of shortest-path algorithms supported by the NS3 simulator under the hood.

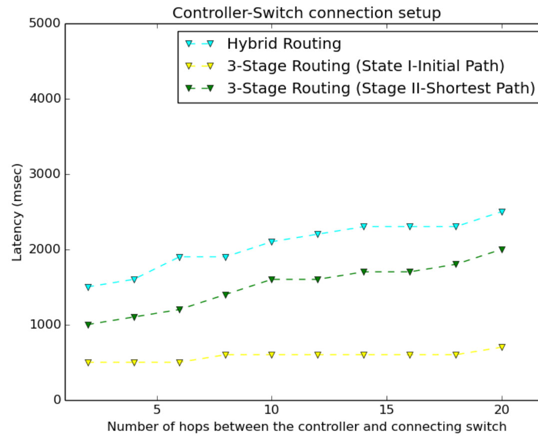

Figure 24: Controller Switch Connection Latency

Figure 23 shows the overview of implementation and various software components used in the implementation.

We have evaluated our approach by comparing its performance with hybrid approach of figure 17. For comparison we used three metric (1) controller-switch connection latency (2) controller-switch reconnection latency and (3) switch-switch connection latency.

In the beginning, controller will try to connect to all the switches using messages OF_Initial_Path_Request and OF_Initial_Path_Response for finding the initial path in stage-I which in turn will be used to install the shortest path in stage-II. We measure the latency to perform stage-I and stage-II and compare it with the hybrid approach. Figure 24 plots this controller-switch connection latency against number of hops between controller and switch. We measure this latency for hyrbid approach, stage-I and stage-II. As can be seen from the figure, our approach basically breaks down the latency required in hybrid approach into two stages (stage I and stage-II). Latency incurred by hybrid approach is approximately sum of latency incurred by stage-I and stage-II. We observed same behavior when we measured the latency for re-connecting controller-switch link during failure. We measured this latency against the number of broken links between controller-switch as seen in the figure 25. Here

also we can see the stage-I and stage-II reconnection latency adds up to the reconnection latency in hybrid approach.

It is evident that our approach breaks up routing into two stages where first stage finds inefficient route to controller but takes lesser time while second stage tries to optimize the route found in first stage but takes more time. This helps overall performance of actual routing between switch-switch connection in stage-III as seen in the figure 26. Figure 26 shows the connection latency among switches against number of hops between them. It is seen clearly that stage-III of our approach outperforms the hybrid approach as number of hops increase between switches. The reason for this result is switch has better connection to controller in our approach than in hybrid approach as shown in figure 24 and figure 25. As in software defined networking, whenever a switch wants to connect to another switch, it requests controller to installs the routing rules. Hence, connection to controller plays a big role in the switch-switch connection latency. In wired networks there is lesser mobility of nodes and hence once controller installs rules in switches, these rules may not need to be changed frequently. Hence routing among switches is not impacted by the controller-switch latency in wired networks. However in wireless mesh networks, as nodes can move more frequently, switch needs to establish a reliable connection to controller in order to improve switch-switch routing.. This is where our approach improves on the hybrid approach. Hybrid approach always tries to find the best route to controller which incurs higher latency and hence controller-switch connection becomes unavailable for longer times. However three-stage approach tries to find the inefficient route to controller incurring smaller latency in stage-I which helps to keep controller-switch connection alive for longer times which helps in reducing the latency in the stage-III.

IV.5 Conclusions

In this chapter, we discussed various challenges in adapting software defined networking paradigm to the wireless mesh networks as SDN is designed to work for wired data

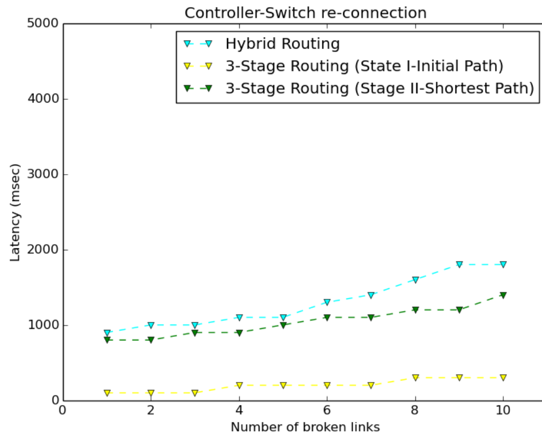

Figure 25: Controller Switch Re-Connection Latency

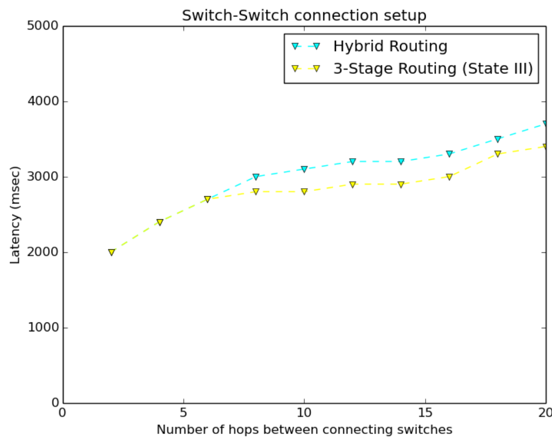

Figure 26: Switch-Switch Connection Latency

center networks. We proposed three-stage routing to efficiently use SDN in wireless mesh networks. In this regard, we proposed extensions to the existing Openflow protocol with three new type of messages which facilitates three stage routing. We then evaluated three-stage routing approach using latency metric one for the connections between controller & switch and another for connection among switches. In the future, we would like to evaluate performance of three-stage routing during the mobility of network nodes including switch and controller.

CHAPTER V

SUMMARY

In this chapter, we describe the summary of our contributions to the field of Software defined networking. We also list our publications related to this work.

V.1 Summary of Contributions

1. InitSDN: we described a solution approach that involves a separate bootstrapping or initialization phase for the SDN network. Our solution is called InitSDN and its architecture involves a number of functionalities that relate to topology, discovery, synchronization, and placement.

2. SDMC: We proposed a novel way of using SDN based multicast (SDMC) for flexible, network-load aware, switch-memory efficient group communication specifically for the data center networks. SDMC efficiently uses combination of unicast and software defined multicast and switches between them at run time agnostic to application and without any additional packet loss to find a better trade off to retain benefits of group communication while avoiding its disadvantages.

3. Openflow based routing in Wireless Mesh Network: We proposed three-stage routing to efficiently use SDN in wireless mesh networks. We described extensions to the existing Openflow protocol with three new type of messages which facilitates three stage routing.

V.2 Summary of Publications

1. Prithviraj Patil, Aniruddha Gokhale, "Towards Reliable Communication in Intelligent Transportation Systems", Poster Proc. of 31st IEEE International Symposium on Reliable Distributed Systems, SRDS 2012 485-486,)

2. Prithviraj Patil, Aniruddha Gokhale, "Maximizing Vehicular Network Connectivity through an Effective Placement of Road Side Units Using Voronoi Diagrams", Poster Proc. Of 13th international Conference on Mobile Data Management, MDM, 2012

3. Prithviraj Patil, Aniruddha Gokhale, "Voronoi-based placement of road-side units to improve dynamic resource management in Vehicular Ad Hoc Networks", International Conference on Collaboration Technologies and Systems, CTS, 2013

4. William Otte, Abhishek Dubey, Subhav Pradhan, Prithviraj Patil, Aniruddha Gokhale, and Gabor Karsai, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment", Proc. of the 16th IEEE Computer Society symposium on object/component/service-oriented realtime distributed computing (ISORC 2013), Paderborn, Germany.

5. Prithviraj Patil, Subhav Pradhan, and Aniruddha Gokhale, "Envisioning a Reusable Framework Based on Dependability Patterns and Principles in Cyber-Physical Systems", 1st International Workshop on Reliable CyberPhysical Systems (WRCPS 2012), Irvine, CA, USA.

6. Prithviraj Patil, Aniruddha Gokhale, Akram Hakiri "Modular and Highly Configurable Computation Mobility Framework for Internet of Things", Institute for Software Integrated Systems (ISIS) Technical Report, ISIS-15-116, 2015, Nashville, TN, USA.

7. Prithviraj Patil, Aniruddha Gokhale, Akram Hakiri "Bootstrapping Software Defined

Network for flexible and dynamic control plane management", IEEE Conference on Network Softwarization (IEEE NetSoft 2015)

8. Prithviraj Patil, Aniruddha Gokhale, Akram Hakiri, Pascal Berthou "Towards a Publish/Subscribe based Open Policy Framework for Proactive Overlay Software Defined Networking", Institute for Software Integrated Systems (ISIS) Technical Report, ISIS-15-115, 2015, Nashville, TN, USA.

9. Prithviraj Patil, Akram Hakiri, Aniruddha Gokhale "Cyber Foraging and Offloading Framework for Internet of Things", The 40th IEEE Computer Society International Conference on Computers, Software & Applications 2016.

10. Yogesh Barve, Prithviraj Patil, Aniruddha Gokhale "A Cloud-based Immersive Learning Environment for Distributed Systems Algorithms", The 40th IEEE Computer Society International Conference on Computers, Software & Applications 2016.

11. Prithviraj Patil, Akram Hakiri, Aniruddha Gokhale, "Adaptive Software-defined Multicast for Efficient Data Center Group Communications", 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN 2016). (Under Review)

12. Akram Hakiri , Prithviraj Patil, Aniruddha Gokhale, "SDN-enabled Wireless Fog Network Management", 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN 2016). (Under Review)

13. Yogesh Barve, Prithviraj Patil, Anirban Bhattacharjee, Aniruddha Gokhale, "A Cloud-based Immersive Learning Environment for Distributed Systems Algorithms", IEEE Transactions on Emerging Topics in Computing (Under Review)

14. Prithviraj Patil, Akram Hakiri, Yogesh Barve, Aniruddha Gokhale, "Openflow Based Three-Stage Routing Protocol for Software Defined Wireless Mesh Networks", IEEE International Conference On Computer Communications 2016 (In Submission)

REFERENCES

[1] Ahmed Abujoda, David Dietrich, Panagiotis Papadimitriou, and Arjuna Sathiaseelan. Software-defined wireless mesh networks for internet access sharing. *Computer Networks*, 93:359–372, 2015.

[2] Ian F Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer networks*, 47(4):445–487, 2005.

[3] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, William Snow, and Guru Parulkar. Openvirtex: A network hypervisor. *Open Networking Summit*, 2014.

[4] Bruno Nunes Astuto, Marc Mendonça, Xuan Nam Nguyen, Katia Obraczka, Thierry Turletti, et al. A survey of software-defined networking: Past, present, and future of programmable networks. 2014.

[5] Roberto Bifulco, Roberto Canonico, Marcus Brunner, Peer Hasselmeyer, and Faisal Mir. A practical experience in designing an openflow controller. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 61–66. IEEE, 2012.

[6] Raffaele Bruno, Marco Conti, and Enrico Gregori. Mesh networks: commodity multihop ad hoc networks. *Communications Magazine, IEEE*, 43(3):123–131, 2005.

[7] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[8] Andrea Detti, Claudio Pisa, Stefano Salsano, and Nicola Blefari-Melazzi. Wireless mesh software defined networks (wmsdn). In *2013 IEEE 9th international conference on wireless and mobile computing, networking and communications (WiMob)*, pages 89–95. IEEE, 2013.

[9] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12. ACM, 2013.

[10] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. Sdn-wise: Design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 513–521. IEEE, 2015.

[11] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on*

Hot topics in software defined networks, pages 19–24. ACM, 2012.

[12] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[13] Linfeng Hong, Dongxu Zhang, Hongxiang Guo, Xiaobin Hong, and Jian Wu. Openflow-based multicast in ip-over-lobs networks: A proof-of-concept demonstration. In *2012 17th Opto-Electronics and Communications Conference*, 2013.

[14] Huawei Huang, Peng Li, Song Guo, and Weihua Zhuang. Software-defined wireless mesh networks: architecture and traffic orchestration. *Network, IEEE*, 29(4):24–30, 2015.

[15] Makoto Imase and Bernard M Waxman. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[16] Aakash Iyer, Praveen Kumar, and Vijay Mann. Avalanche: Data center multicast using software defined networking. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*, pages 1–8. IEEE, 2014.

[17] Y Jarraya, T Madi, and M Debbabi. A survey and a layered taxonomy of software-defined networking. 2014.

[18] Anand Krishnamurthy, Shoban P Chandrabose, and Aaron Gember-Jacobson. Pratyaastha: an efficient elastic distributed sdn control plane. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 133–138. ACM, 2014.

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

[20] Dan Li, Henggang Cui, Yan Hu, Yong Xia, and Xin Wang. Scalable data center multicast using multi-class bloom filter. In *Network Protocols (ICNP), 2011 19th IEEE International Conference on*, pages 266–275. IEEE, 2011.

[21] Dan Li, Mingwei Xu, Ming-chen Zhao, Chuanxiong Guo, Yongguang Zhang, and Min-you Wu. Rdcm: Reliable data center multicast. In *INFOCOM, 2011 Proceedings IEEE*, pages 56–60. IEEE, 2011.

[22] Yuting Liu, Cheng Li, Ramachandran Venkatesan, et al. Wireless mesh networks: A survey. 2007.

[23] Cesar AC Marcondes, Tiago PC Santos, Arthur P Godoy, Caio C Viel, and Cesar AC Teixeira. Castflow: Clean-slate multicast approach using in-advance path processing in programmable networks. In *Computers and Communications (ISCC), 2012 IEEE*

Symposium on, pages 000094–000101. IEEE, 2012.

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

[25] Yuhuai Peng, Lei Guo, QingXu Deng, Zhaolong Ning, and Lingbing Zhang. A novel hybrid routing forwarding algorithm in sdn enabled wireless mesh networks. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 1806–1811. IEEE, 2015.

[26] Python. Fabric: Ssh for application deployments. `http://www.fabfile.org/`, January 2015.

[27] Y Reddy, D Krishnaswamy, and BS Manoj. Cross-layer switch handover in software defined wireless networks. In *2015 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*, pages 1–6. IEEE, 2015.

[28] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *USENIX; login*, 38(5):128–134, 2013.

[29] Stefano Salsano, Giuseppe Siracusano, Andrea Detti, Claudio Pisa, Pier Luigi Ventre, and Nicola Blefari-Melazzi. Controller selection in a wireless mesh sdn under network partitioning and merging scenarios. *arXiv preprint arXiv:1406.2470*, 2014.

[30] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[31] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association, 2010.

[32] Stanford University. Network topology (internet2). `http://snap.stanford.edu/data/#p2p`, January 2015.

[33] Stanford University. Network topology (p2p). `http://snap.stanford.edu/data/#p2p`, January 2015.

[34] Di Wu, Dmitri I Arkhipov, Eskindir Asmare, Zhijing Qin, and Julie A McCann. Ubiflow: Mobility management in urban-scale software defined iot. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 208–216. IEEE, 2015.

[35] Hanjie Yang, Bing Chen, and Ping Fu. Openflow-based load balancing for wireless mesh network. In *Cloud Computing and Security*, pages 368–379. Springer, 2015.

[36] Guang Yao, Jun Bi, Yuliang Li, and Luyi Guo. On the capacitated controller placement problem in software defined networks. 2014.

[37] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 41(4):351–362, 2011.

[38] Yang Yu, Qin Zhen, Li Xin, and Chen Shanzhi. Ofm: A novel multicast mechanism based on openflow. *Advances in Information Sciences & Service Sciences*, 4(9), 2012.